AFRL-IF-WP-TR-2001-1511

RECONFIGURABLE AND ADAPTIVE COMPUTING ENVIRONMENTS

Dr. Dinesh Bhatia

University of Cincinnati
Office of Sponsored Programs
P.O. Box 210627
Cincinnati, OH 45221-0627

March 2000

FINAL REPORT FOR PERIOD 07 AUGUST 1996 – 08 AUGUST 1999

INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334

NOTICE

[illegible]

KERRY L. HILL
Project Engineer
Hardware Team
Embedded Info Sys Engineering Branch

[illegible]

ALFRED J. SCARPELLI
Team Leader
Hardware Team
Embedded Info Sys Engineering Branch

[illegible]

JAMES S. WILLIAMSON
Chief
Embedded Info Sys Engineering Branch

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 074-0188*

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** March 2000	**3. REPORT TYPE AND DATES COVERED** Final Report, 08/07/1996 – 08/08/1999

4. TITLE AND SUBTITLE Reconfigurable and Adaptive Computing Environments	**5. FUNDING NUMBERS** C: F33615-96-C-1912 PE: 62204F PR: 6096 TA: 40 WU: 34
6. AUTHOR(S) Dr. Dinesh Bhatia	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Cincinnati Office of Sponsored Programs P.O. Box 210627 Cincinnati, OH 45221-0627	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) INFORMATION DIRECTORATE AIR FORCE RESEARCH LABORATORY AIR FORCE MATERIEL COMMAND WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334 POC: Kerry L. Hill, AFRL/IFTA, 937-255-7698 x3604	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER** AFRL-IF-WP-TR-2001-1511

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(Maximum 200 Words)***

This Reconfigurable Computing (RC) or Adaptive Computing System (ACS) program focused on the development of both reconfigurable computing platforms and the associated programming support environments to demonstrate the viability of RC. This was demonstrated by exploring the ability to program RCs in a main/integrated C application program and by investigating new, partially reconfigurable technology. A Xilinx 4000 Field Programmable Gate Array (FPGA) series board and a Xilinx 6200 FPGA-based board were developed as part of this effort. The C compiler technology was developed more for a hardware pragma-based implementation, which leveraged hardware macro libraries and worked quite effectively. The Xilinx 6200 board was interesting from the standpoint that the 6200 FPGAs are partially reconfigurable. The shortfalls of this product family include poor chip design/manufacture, resulting in the inability to utilize a good portion of the FPGA logic resources. Another shortfall is a lack of functional programming tools. In spite of these problems, the team was able to exercise the partial reconfigurability of the devices by developing programming tools of their own.

14. SUBJECT TERMS	**15. NUMBER OF PAGES** 88
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** SAR

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

Chapter 1

Introduction to Reconfigurable Computing

Computing paradigms are constantly evolving. Since the creation of ENIAC in 1946, computers have been built around the notion of a microprocessor. In following years, other computer models like stack and memory machines, vector processors, parallel systems, and distributed system have all been devised to increase computational ability and speed. Recently, a new computing paradigm has emerged called *reconfigurable computing.*

Reconfigurable computing is thought of in many different ways; however, each viewpoint is a variation of a basic concept. *Reconfigurable computing is essentially performing any type of computing on reconfigurable hardware.* Reconfigurable computing, termed *RC* here, primarily focuses on hardware that has the ability to be *repeatedly* re-programmed; nevertheless, systems with once-programmable hardware could also be classified as a type of *RC*. Another way of considering *RC* is the ability to implement software in hardware. A clear distinction is not always made between implementing software *on hardware* as is the case with a microprocessor because some reconfigurable systems are in fact reconfigurable processors. Usually, *RC* is considered in terms of implementing software *in hardware*; i.e., mapping a software program or algorithm into a hardware logic gate-level implementation. Implementing processors in reconfigurable hardware is often very slow compared to custom ASICs; however, direct logic mapping of a program onto reconfigurable hardware often results in tremendous speed-ups over a conventional microprocessor.

1.1 Field-Programmable Gate Arrays

Reconfigurable computing is made possible through the advent of FPGAs, or **F**ield-**P**rogrammable **G**ate **A**rrays. FPGAs are arrays of programmable logic, including general purpose registers, multiplexers, tristates, and other forms of optimized logic. In many regards, FPGAs are similar to PLAs (programmable logic arrays) or EPLD (erasable- programmable logic devices); however, FPGAs contain more types of logic and have several types of programmable interconnects and input-output blocks. Mask programmable gate arrays usually have more available logic than FPGAs and can be clocked at much higher speeds; however, the flexibility, programmability, and low cost of FPGAs make them an ideal choice for most applications.

FPGAs come in different types of architectures and packages. The most prevalent architecture is the array of logic blocks. Figure 1.1 shows an example of an array architecture made by Xilinx [25].

The logic block in an FPGA is often called a *CLB*, or *configurable logic block.* Another term for the logic block is a *PFU*, or *programmable function unit.* The CLBs utilize look-up tables to implement different input limited functions. In a Xilinx FPGA, each CLB has several function generators (each a separate

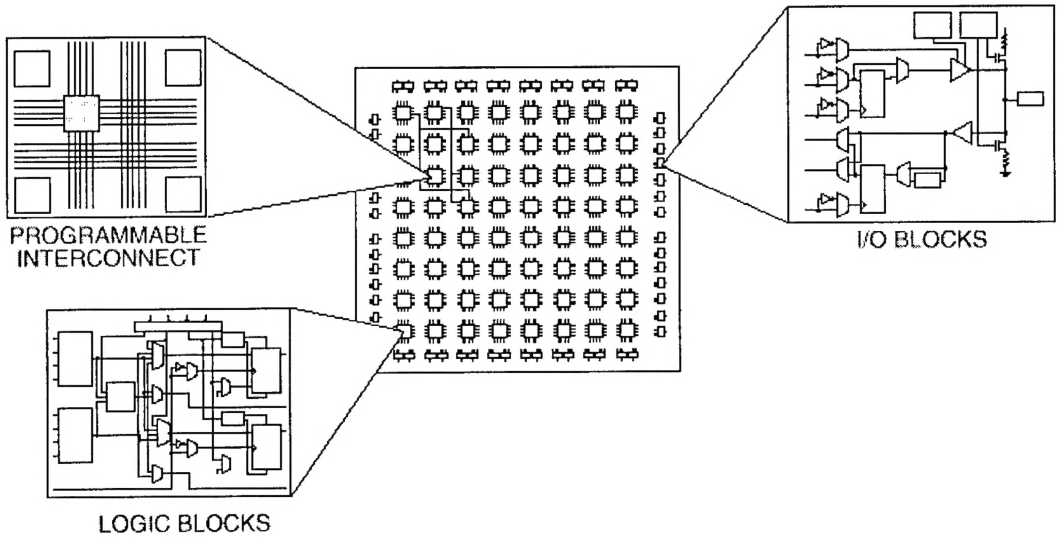


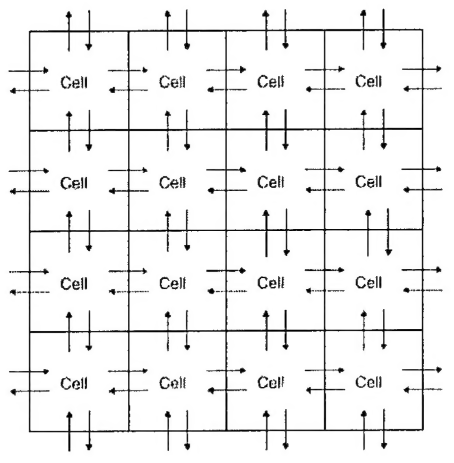
Figure 1.1: *An example of a typical Xilinx FPGA (courtesy of Xilinx).*

look-up table) that can be connected together to produce larger functions [24]. Likewise, using the interconnects between the CLBs, very large functions can be created by connecting a number of CLBs together. Technology allows manufacturers to place more and more CLBs on a chip, increasing the number of logic gates available. The greater the amount of logic gates on an FPGA, the more realizable reconfigurable computing becomes.

Another FPGA architecture, called the sea-of-gates, is also very popular. Vendors such as Atmel[4] and Xilinx[27] use sea-of-gates architectures to provide very fine-grained reprogrammability and high clocking speeds. Figure 1.2 shows two examples of sea-of-gates architectures.

The sea-of-gates is comprised of single cells that use a small look-up table to implement 2 to 3 input functions. In combination with many other cells, these cells become a sea-of-gates with the ability to yield a large numbers of logic gates. Since a sea-of-gates resembles a memory structure, configuration and accessing the FPGA is much the same as reading or writing to memory. For example, the Xilinx XC6200 series FPGA has both an address and data bus that allows both reads and writes into individual cells or configuration bits. Because of this capability, these types of FPGAs interface nicely with microprocessors and bus-type of interconnections. In addition, the ability to program individual cells allows for *partial reconfiguration* of an FPGA; whereas, FPGAs with array structures usually have to be programmed all at once in a bit-serial fashion.

Several other FPGA architectures also exist. Some FPGAs are essentially EPLDs or PLAs; whereas, others include analog components to create mixed-signal FPGAs [13][28]. Many of the commercial applications for FPGAs use one-time programmable FPGAs as with the popular MACH series FPGAs from AMD[1]. One-time programmable FPGAs use anti-fuses to "burn" the configuration into an FPGA. However, for reconfigurable computing to take place, SRAM-based FPGAs with the capability of being reprogrammed many number of times need to be used. SRAM-based FPGAs use SRAM memory cells to store the configuration of the FPGA. Each configuration block and interconnect switch-block are configured based on the values stored in the SRAM. The extra configuration logic reduces the available amounts of usable logic as compared to anti-fuse FPGAs, but the added flexibility, and fast reconfigurability make



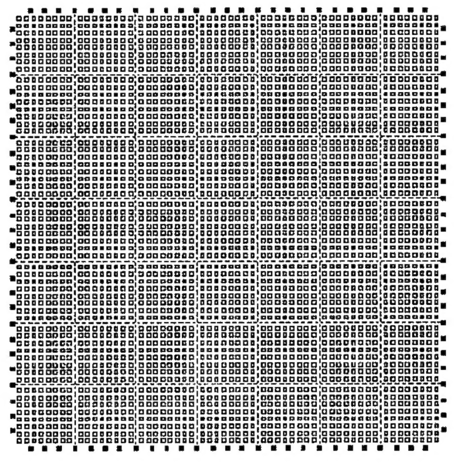


Figure 1.2: *Two examples of sea-of-gates FPGA architectures. On the left, a Xilinx XC6200 series, and on the right, an Atmel AT6000 series (courtesy of Xilinx and Atmel).*

SRAM-based FPGAs suitable for reconfigurable computing.

1.2 Motivation for Reconfigurable Computing

With the speed of microprocessors growing faster, why would reconfigurable computing be used? The answer is because of its speed-up over implementing software on a microprocessor. However, many would argue that the speeds of the Intel Pentium microprocessors clocking above 500MHz and the DEC Alpha microprocessors clocking up to 600MHz makes reconfigurable computing neither worthwhile nor capable of outperforming custom microprocessors.

In actuality, any program or algorithm implemented in combinatorial logic (gate-level) is almost always going to outperform a sequential execution on a microprocessor. In 1978, Rauscher and Agrawala[21] presented work that showed if a program was profiled and its repetitive machine code was added to the processor's microinstructions, then the program would execute much faster on the new architecture. In essence, their research showed that if the computer hardware can be customized for each new application, then the application will execute much faster on the new custom hardware. An example of such a machine is the CM-2X[9]. The CM-2X is a CM-2 parallel computer with a Xilinx XC4005 FPGA instead of a Weitek 3364 coprocessor. The CM-2X showed a factor of 4 speed-up over a CM-2 with a Weitek coprocessor and a factor of 6 speed-up over a CM-2 without any coprocessor. *RC* also allows the possibility of *adaptive* computing; i.e., the hardware can **adapt** or reconfigure in response to external stimuli. Similarly, adaptability allows *fault-tolerant* systems to automatically reprogram themselves by moving logic out of faulty circuitry into properly functioning circuitry. Each of these benefits make *RC* an attractive form of computing.

Since *RC* demonstrates such a noticeable speed-up over microprocessors, another question to ask is, "What types of applications are suited for reconfigurable computing?" Essentially, *RC* can be used for any type of application. Figure 1.3 illustrates some of the possible ways current *RC* systems are being used, and the hardware architectures that have been developed to implement those applications.

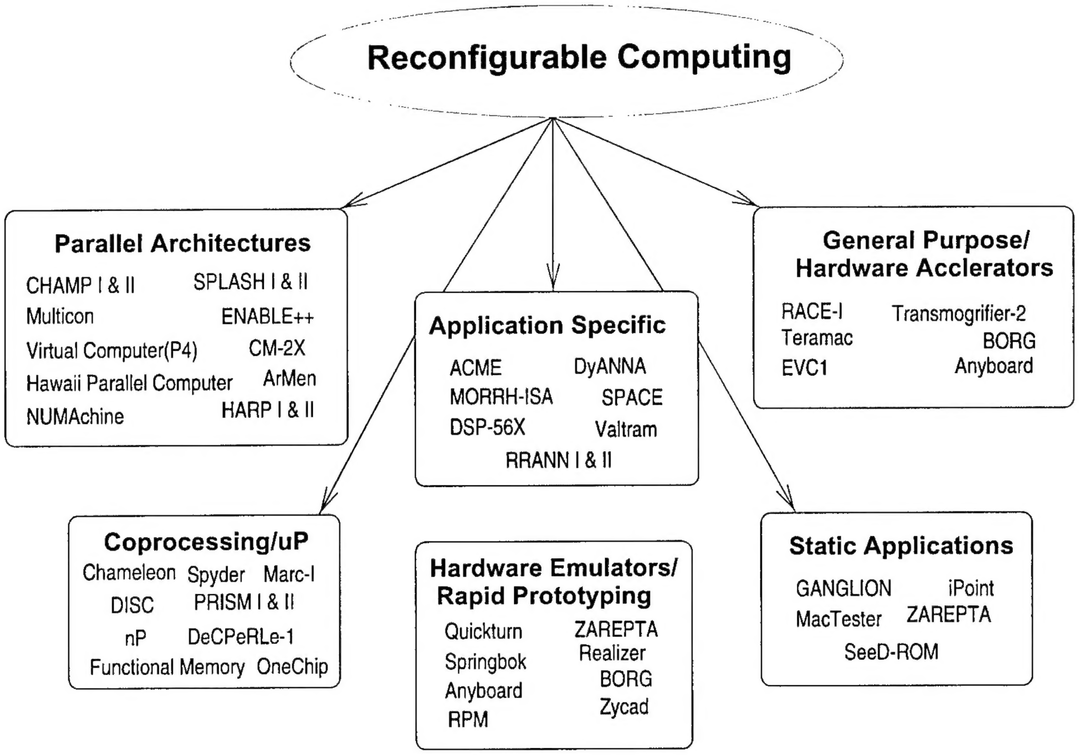


Figure 1.3: *Applications of reconfigurable computing and examples of hardware systems that perform that type of* RC. *Since the systems are reconfigurable, most of them can be used for several types of applications.*

1.2.1 Coprocessing

As shown in Figure 1.3, many systems are designed specifically for coprocessing. Coprocessing usually appears in two forms: 1) the coprocessor is implemented as some type of microprocessor, or 2) the coprocessor is made of several FPGAs with uncommitted logic for general purpose applications. FPGA-based microprocessors clock slower than custom ASICs; however, since their functionality and op-codes can be modified for the application at hand, their overall performance can be much better than a general-purpose microprocessor. The second group of coprocessing systems are what is typically thought of as reconfigurable computers. A system, usually with some interface to a host, has multiple FPGAs that can implement any algorithm or function that is mapped onto the FPGAs' logic. In fact, most FPGA-based systems can be classified to some degree as coprocessors since almost all systems interface with a host. Generally, applications are described in either a HDL (such as VHDL), C, or C++, and then compiled or synthesized into a bitstream configuration for the FPGAs on the *RC* system. Some examples include the Anyboard [10], BORG [6][7], PRISM [3] [23], the Virtual Computer (P4 and EVC1) [5], Transmogrifier [12], ENABLE++ [16], and RACE [22].

1.2.2 Application Specific Computing

Reconfigurable computers are often designed to be optimized for a particular task in an effort to outperform the corresponding software implementation. Custom DSP ASICs are very fast, but also general purpose. *RC* systems allow for custom configuration of the hardware for a particular type of filter or algorithm, usually resulting in a noticeable speed-up over DSP chips. Similarly, implementing neural networks in *RC* systems is very common. Neural networks use several levels of nodes, and execution in software can be extremely slow. Consequently, reconfigurable computers are often designed such that each FPGA represents a node of the neural network.

1.3 Types of Reconfigurability

Reconfigurable computers can be broadly classified as having one of two types of reconfigurability—*static and dynamic reconfigurability.* These categories are constantly changing since the definitions of *static* and *dynamic* reconfigurability change from year to year, depending on current research and hardware developments. In this discussion, "statically reconfigurable" refers to having the ability to reconfigure a system, but once programmed, its configuration remains primarily static. In contrast, any *RC* system that is constantly reconfigured is considered "dynamic". At first, it seems contradictory to call something static that is *reconfigurable* since *reconfigurable* implies that it does not maintain a static configuration; however, the term "static" refers more to the *RC's* mode of operation. In fact, these definitions can be used rather loosely since there is overlap between two. Figure 1.4 shows different categories of reconfigurability for *RCs*.

1.3.1 Static RCs

Static *RCs* are useful primarily for specific tasks. Even though most *RCs* are considered dynamic, several static *RCs* exist. For example, GANGLION [8] is a connectionist classifier used in manufacturing for identifying edges and objects. Static *RCs* include the MacTester [11], iPoint [18], and SeeD-ROM [20].

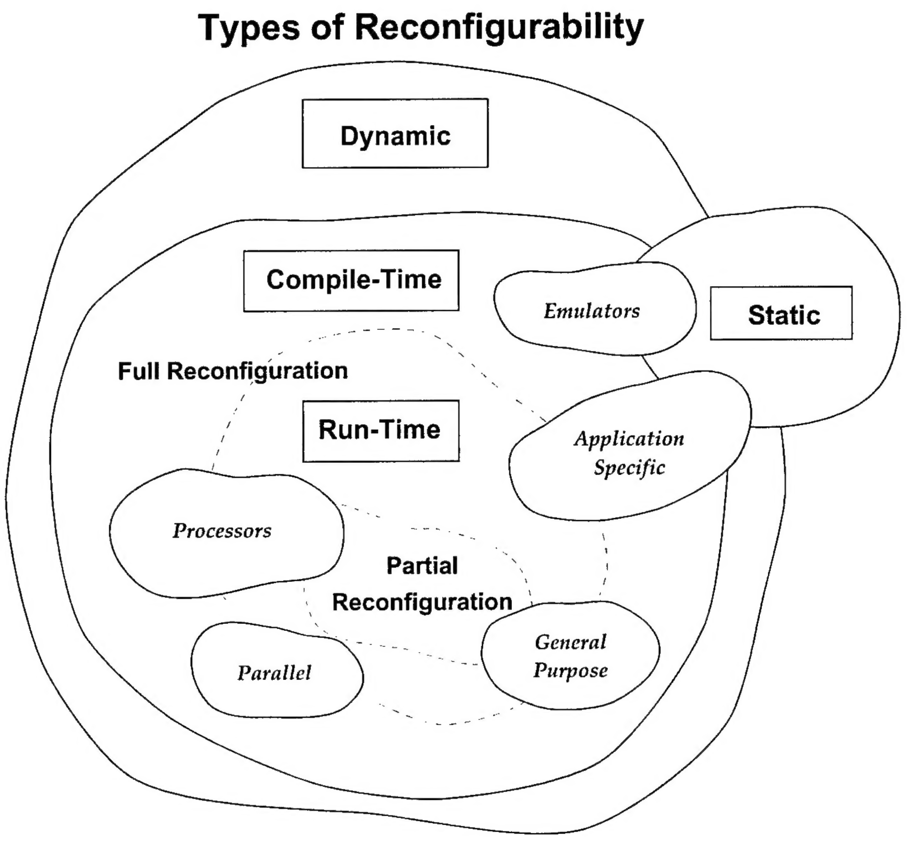


Figure 1.4: *Possible types of reconfigurability a reconfigurable computer can have.*

1.3.2 Dynamic RCs

The definition of *dynamically reconfigurable* is as changing as the term itself. When SRAM-based FPGAs became available, the term *dynamically reconfigurable* referred to the ability to be able to simply reconfigure hardware in-circuit. Over time, it has come to mean "run-time" reconfigurable, and most recently, "partially" reconfigurable. In the broadest sense of the term, all systems with the capability of being reconfigured in-circuit are "dynamically" reconfigurable. However, distinctions can be made between different types of dynamically reconfigurable systems. For example, most systems can be considered as *compile-time* reconfigurable[17]. In other words, the hardware configuration for an application is determined at compile-time and does not change for the duration of the application [15][14]. Systems that change their hardware configuration during the execution time of the application can be thought of as *run-time* reconfigurable .

Subdividing run-time reconfigurable systems even further, a distinction can be made between *RCs* that require all their FPGAs to be *fully* reconfigured or can be *partially* reconfigured. Partial reconfigurability has also been termed as *local* run-time reconfigurability [17]. All Xilinx FPGAs, except for the new XC6200 series, require the whole chip to be reconfigured any time the configuration needs to be modified. Since most *RCs* are based on Xilinx FPGAs, most *RCs* are not partially reconfigurable. Partially reconfigurable FPGAs are continuing to become more common place, like the new VIRTEX FPGA architecture from Xilinx. Using partially reconfigurable FPGAs, *RCs* can also become *adaptive* and *fault tolerant.* In other words, suppose a *RC* is programmed with an application that receives feed-back that modifies its behavior as in the case of a neural net. Instead of reconfiguring each entire FPGA, only subtle and minute changes could be made to the FPGA while the application is still executing; thus, the *RC* can be consider adaptive. Likewise, processors implemented on partially reconfigurable FPGAs could slightly modify their opcodes or functionality by making only partial chip reconfigurations.

1.4 Obstacles of Reconfigurable Computing

Reconfigurable computing can result in some large speedups over software execution; however, great obstacles still must be overcome for it to have widespread use. The largest obstacles are the *overhead* costs associated with *RC*, and the programming difficulty. The overhead can be thought of in terms of three components: 1) the time it take to perform any data transfers, $t_{Data\ Transfers}$; 2) the time required by the operating system to perform the data transfers and control of the *RC*, $t_{OS\ Cost}$; and 3) the time required to fully configure an FPGA with an application, $t_{Config\ Time}$. In other words,

$$t_{Overhead} \approx t_{Data\ Transfers} + t_{OS\ Cost} + t_{Config\ Time} \tag{1.1}$$

For most *RC* systems, $t_{Config\ Time}$ is constant (except in the case of partial reconfiguration). The $t_{OS\ Cost}$ varies depending on the amount of file and device driver accesses made. In fact, the OS cost can be determined once the individual time is known for a particular type of OS access. In other words, the OS cost can be approximated in the following way,

$$t_{OS\ Cost} \approx n \times \bar{t}_{File\ Access} + m \times \bar{t}_{Driver\ Access} \tag{1.2}$$

where *n* is the number of files accesses and *m* is the number of driver accesses. This equation will give only a rough approximation since the size of the files accessed may vary greatly. If a program requires some sort of data file input, then whether it is implemented in software or in hardware, there will still be the same OS overhead cost of reading in the data. Therefore, the cost of reading in a data file should not be considered as an overhead cost of reconfiguration computing, rather simply a cost of any type of computing. Instead, only the cost of reading in the hardware configuration should be considered. Since most systems must be fully reconfigured at one time, the amount of configuration data will remain constant and the OS overhead associated with reading the data will also remain basically constant. Likewise, the number of driver accesses for configuring an *RC* will remain essentially the same. The only variable would be the number of driver accesses needed for transferring data. In other words,

$$t_{OS\ Cost} \approx (C_{Config-File\ Accesses} + C_{Config-Driver\ Accesses}) + m \times \bar{t}_{Data-Driver\ Access} \tag{1.3}$$

where $C_{Config-File\ Accesses}$ is a constant time required to read in the configuration data, $C_{Config-Driver\ Accesses}$ is a constant time required to access the device driver to configure the *RC*, *m* is the number of data transfers, and $\bar{t}_{Data-Driver\ Access}$ is the average time required to access the driver to perform a data transfer.

The largest cost incurred with reconfigurable computing is usually the cost associated with transferring the data into the *RC*. With software, once the data is read into memory, it can immediately start using the data; whereas, with an *RC*, the data has to be transferred to the system, which can incur a high latency. If an *RC* sits on the computer's main bus as many coprocessing *RCs* do, then this data transfer overhead can be eliminated. Consequently, equation 1.1 can be re-written as

$$t_{Overhead} \approx t_{Data\ Transfers} + m \times \bar{t}_{Data-Driver\ Access} + C_{Config\ Time} \tag{1.4}$$

where $t_{Data\ Transfers}$ depends on the data size, m is the number of device driver accesses for the data transfers, $\bar{t}_{Data-Driver\ Access}$ is the average amount of time required to make a driver access for transferring the data, and $C_{Config\ Time}$ is the sum constant time associated with programming the *RC*, including the file system and driver accessing time.

If the overhead cost of reconfigurable computing outweighs the time required to implement an application in software, then little reason exists to do reconfigurable computing. In fact, even if the time to implement in software is equal to the time necessary for hardware, then it is still easier to execute in software. The difficultly of designing an application in hardware greatly outweighs the difficulty of writing software. Consequently, to make reconfigurable computing attractive as a viable solution, the following must hold true,

$$t_{software} \gg t_{hardware} + t_{overhead} \tag{1.5}$$

where $t_{software}$ is the time required to execute an application in software, $t_{hardware}$ is the time required to execute the same application in hardware, and $t_{overhead}$ is the overhead cost given in equation 1.4 for the application. Since the overhead is dependent on the amount of data required for an application, applications with large data sets may result in too large of an overhead cost for reconfigurable computing. Likewise, even if $t_{software} \leq t_{hardware}$, the overhead cost will usually make executing in software faster. Consequently, trying to execute everything on an *RC* may not be worth it; instead, *RCs* are better suited for very time-consuming applications where $t_{software} \gg t_{hardware}$, or as coprocessing systems for software-hardware co-execution.

Another obstacle to reconfigurable computing is the difficulty of mapping software constructs and algorithms into raw logic gates. In fact, behavior synthesis can be a very hard problem for most applications. Likewise, all *RCs* have memory and logic limitations which make designing the applications even more difficult. For example, implementing floating point operations in hardware requires a large number of bits to provide adequate precision for the numbers. If an *RC* does not have enough memory or the data bus size is too small, then a resulting slow down will occur from more memory fetches. Likewise, floating point operations can require a lot of logic that may not be available on the *RC*. Techniques like *temporally partitioning*, i.e., slicing a hardware design into multiple stages or time slices, can help overcome some of the *RC's* hardware limitations. Lastly, designing hardware applications often requires some sort of hardware knowledge and experience that an average software programmer may not have.

Chapter 2

Hardware Platforms for Reconfigurable Computing

The Reconfigurable Computing Environments contract resulted in several accomplishments. These include the Hardware platforms named RACE-I and the NEBULA architectures. RACE-I is built around Xilinx 4000 family parts and suppoorts static and dynamic runtime reconfigurability. NEBULA architecture was built around Xilinx 6200 family of parts and supports all features supported by RACE-I as well as the partial reconfigurability. In this chapter, we describe the hardware features for RACE-I and the NEBULA architecture.

2.1 The RACE-I Architecture

The RACE architecture consists of a host workstation, an interface, and an external board of FPGAs, which is used for co-processing applications. The following sections describe in detail each component of the RACE system.

2.2 Workstation

The workstation that is used to interface with RACE is a Sun Sparc IPC workstation running SunOS 4.1.3. The workstation's internal bus uses the SBus protocol, which is an industry open standard. The SBus can transfer data up to 16, 32-bit word bursts or double-word bursts of 64-bits (extended mode). In a Sparcstation, the SBus' frequency is 25MHz. Ideally, the SBus can transfer a word per clock, which equals to 100MB per second (168MB in extended mode). In practice on an IPC, typical bus transfer rates are about 25MB per second since it can only handle up to 4 word bursts.

2.3 DPS-1 Interface

The interface between the RACE co-processing architecture and the workstation is a DAWN VME Products DPS-1 DMA SBus prototyping board. The DAWN board has an LSI Logic L64853A Enhanced SBus DMA Controller [19] as well as an area for wire-wrap prototyping. The LSI L64853A DMA controller has two modes of operation. The first mode is *master* mode. In master mode, the L64853A performs DMA

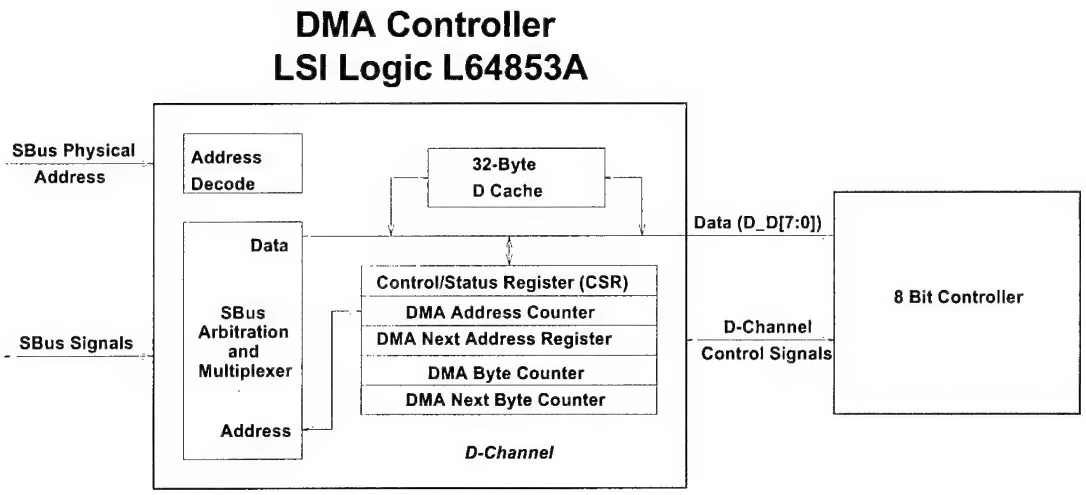


Figure 2.1: *The LSI Logic L64853A DMA controller. This figure illustrates the five registers used during a DMA transfer, and the D-channel with its data bus and control signals.*

(*direct memory access*) transfers independently of the workstation's CPU. The L64853A requests access to the SBus and supplies the virtual memory address when granted access to the bus. Only one master or slave access can occur at any one time on the SBus; however, the CPU can continue to execute while simultaneous memory transfers are occurring on the SBus. The second mode of operation of the L64853A controller is *slave* mode. In slave mode, the CPU acts as the SBus master and supplies the memory address for access into the memory map of the SBus card. This is analogous to "peeking" and "poking" directly into the memory. Slave accesses are easily made from a high level language because once the SBus card's memory map is mapped into the system's memory, the user program can read and write directly to the memory address of the SBus card. Since the CPU acts as the SBus master during a slave access, the CPU is busy performing the data transfer instead of other system or user programs. During either slave or master mode accesses, the L64853A acts as a conduit, passing data between the SBus and whatever is connected to its input/output channel. An internal 32-byte cache helps reduce the amount of necessary SBus accesses. Figure 2.1 illustrates the LSI L64853A DMA controller. For more a detailed description of the L64853A, refer to the LSI *L64853A Enhanced SBus DMA Controller Technical Manual* [19]

2.4 RACE Board

The RACE board can be divided into three parts: 1) the controller, 2) the memory, and 3) the FPGAs (Figure 2.2, which illustrates the RACE board). In addition to these three components of the RACE board, the bus, clocking, and power components will also be described in this section.

2.4.1 RACE Controller

The RACE controller is perhaps the most critical part of the RACE system since it determines the functionality of the board. The controller is made from a Xilinx XC4013E FPGA with a 208-pin surface mount package. In an effort to make RACE totally reconfigurable for reconfigurable computing, the controller does

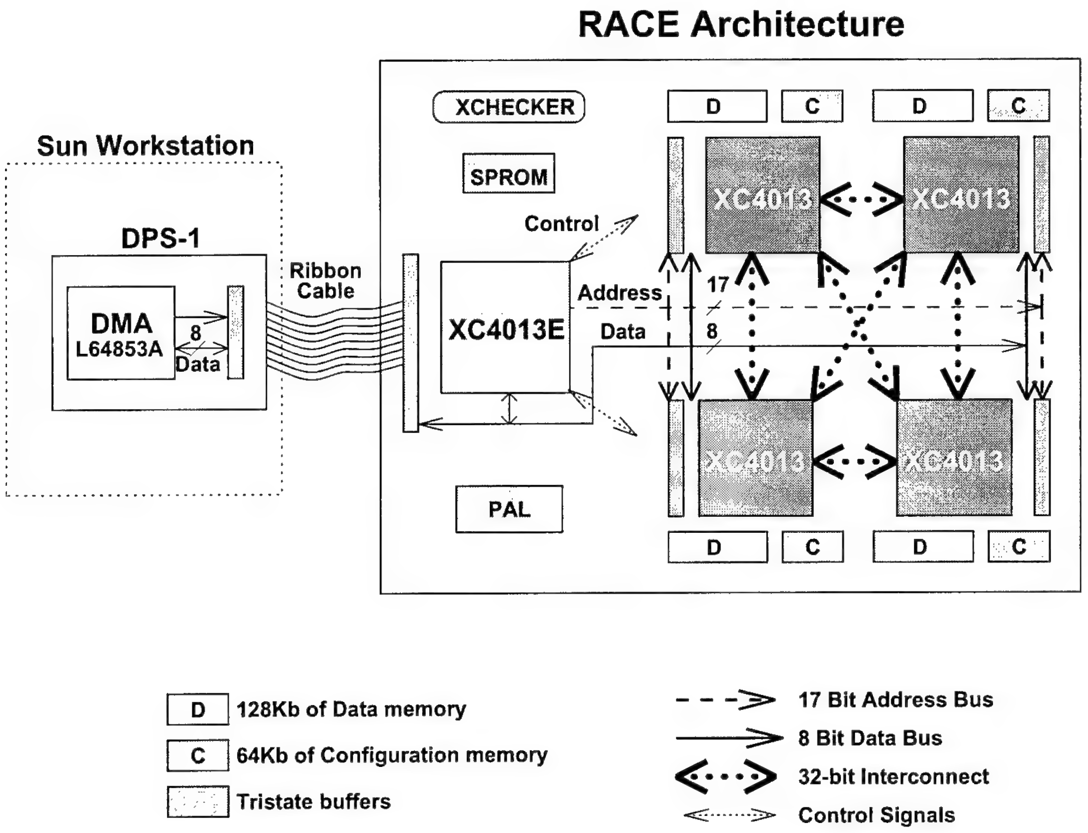


Figure 2.2: *The RACE architecture. The architecture consists of a host and interface, a Xilinx XC4013E controller, four Xilinx XC4013 FPGAs, 512KB SRAM used for function data, and 256KB SRAM used for configuration data.*

not have to have a static configuration, but can be reconfigured with optimized configurations for particular applications. In general for most applications, the controller's configuration need not change. The controller's functions include: 1) DMA transfers (which includes generating the appropriate addresses for the address bus), 2) programming of the FPGAs on the RACE system, 3) control of the functions implemented on the RACE system, 4) control of the on-board memory and memory arbitration, and 5) communicating the resulting value of functions through polling or interrupts.

Since the controller is reprogrammable, there is no set way the controller has to behave, except to have the aforementioned five types of functionality. The following describes the default configuration for the controller developed for RACE and how it is used. The RACE controller consists of four primary registers, address decode logic, several state machines, and other control logic. Figure 2.3 illustrates the internal structure of the RACE controller. For applications that need a modified controller, a basic controller interface (static) can be used in a schematic and interfaced with an application specific (user-definable) portion.

The four registers of the RACE controller are: 1) the **Control** register (CR), 2) the **Flag** register, 3) the **Address** register, and 4) the **Count** register. Since the L64853A has only an 8-bit D-channel, the registers are split into two or three individual registers so that values larger that 8-bits can be stored. For example, the CR and flag register are 16-bits wide; therefore, two read/write accesses have to be made to fully access

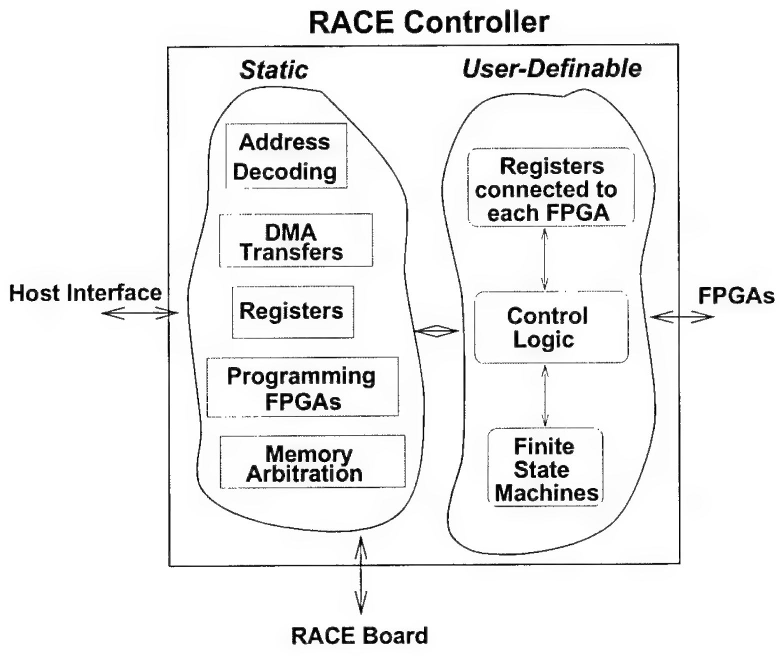


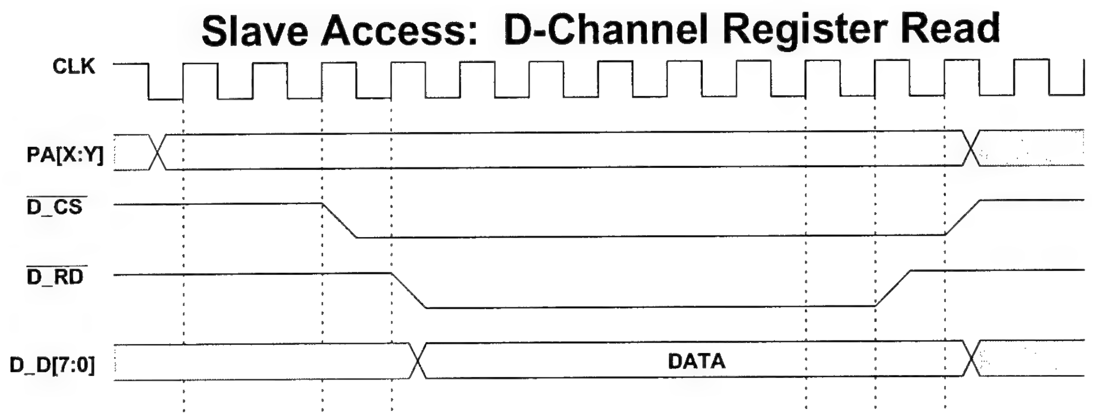
Figure 2.3: *The RACE controller. The static portion (left) can be interfaced with a user-definable (right) portion.*

each register. The address and count registers consist of three bytes because they need to be able to access 128KBytes (17-bits) of memory that is local to each FPGA.

Slave Accesses

Accessing the registers inside the controller involves making SBus *slave* accesses to the RACE board. During a slave access, the microprocessor in the workstation is performing the reading or writing into memory. The microprocessor, governed by a user program or kernel device driver, supplies the appropriate address for the SBus card that it wants to access. Each slot on the SBus has a designated virtual address that gets mapped into the kernel memory. Depending on the address decode logic on each SBus card, this determines what actually gets accessed during a slave data transfer. For example, the L64853A on the DPS-1 interface has a port address of 0x800000 for its CSR. Therefore, to write a value to the CSR of the L64853A, the user program or kernel device driver would write to the following virtual address: SBus slot virtual address + 0x800000. The physical address would be placed on the SBus' 28-bit physical address lines and the address decode logic in the L64853A would recognize the access as a write into its CSR register. Likewise, if the appropriate physical address is supplied on the SBus, then accesses can be made into the internal registers of the RACE controller.

During a slave access, the L64853A basically provides the needed SBus protocol interface and buffers the data being sent, but is essentially by-passed by the microprocessor (which is acting as the SBus DMA master). When the microprocessor (or any other DMA master) places a slave request on the SBus, the L64853A asserts the chip select signal, $\overline{D\_CS}$, to the D-channel controller. Then, either the $\overline{D\_RD}$ or $\overline{D\_WR}$ signal is asserted depending on the direction of the data transfer as shown in Figure 2.4. The D-



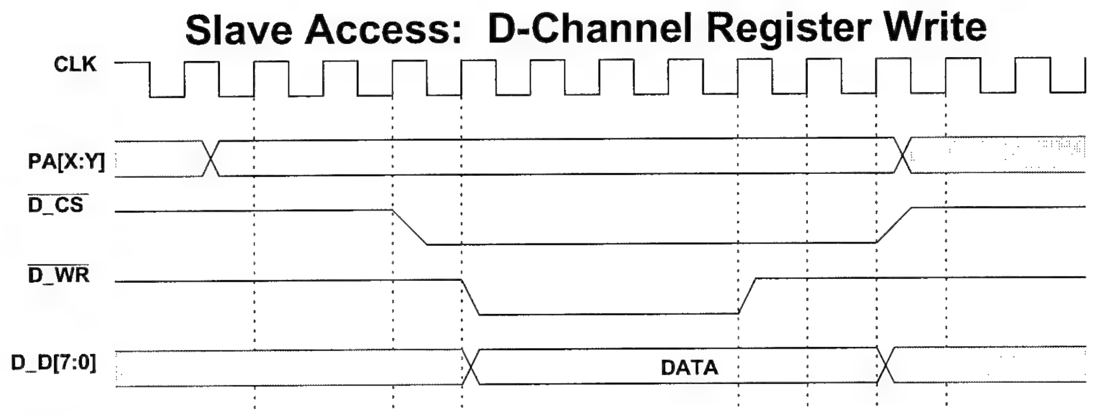


Figure 2.4: *L64853A D-Channel signals during a slave access with the D-Channel controller.*

channel reads or writes the 8-bit data at the memory location specified by the SBus' physical address lines.

Control Register

The **Control** register (CR) is used control the functionality of the controller. For example, if a user wants to program a particular FPGA, then the appropriate value is written to the CR, and the controller performs the programming of that FPGA. Figure 2.5 illustrates the individual bits of the CR, and Table 2.1 describes the purpose of each bit. All of the CR's bits are read/writable except for the error bit, which is set when a programming error occurs. Since only 8-bit data transfers can occur at one time through the LSI L64853A's D-channel, the CR is split into two registers—CR1 and CR2.

Address and Count Registers

The **Address** and **Count** registers are used during DMA transfers to provide the memory addresses and number of bytes to be transferred, respectively. Since each FPGA on the RACE board has up to 128KB of local memory for data, 17 bits are needed to fully address each memory location. Therefore, the address and

Control1 (Bits 0-7)			
Bit	Function	Enable Value	Description
0	**Reset**	RESET=1 *(R/W)*	Resets the entire RACE system. Resets the controller and erases the configuration of the FPGAs specified by bits 8-11.
1	**Interrupts Enabled**	ENABLED=1 *(R/W)*	Enables the controller to assert its $\overline{D\_IRQ}$ signal upon completion of a function's execution or programming of an FPGA.
2	**DMA On**	ON=1 *(R/W)*	Starts the DMA transfer of data. Once a DMA transfer is finished, this bit is reset.
3	**Program**	ON=1 *(R/W)*	Begin programming of FPGA(s). Once this bit is set, the FPGAs enabled in bits 8-11 are programmed with the bitstream configuration contained in the bank of configuration memory specified by bits 12-15.
4	**Bitstream**	ON=1 *(R/W)*	Turning this bit on causes any subsequent DMA transfers to transfer data to or from the configuration memory.
5	**Clock_On**	ON=1 *(R/W)*	Turns the 8MHz CCLK clock used to configure the FPGAs in *Synchronous Peripheral* mode.
6	**Error**	ERROR=1 *(Read Only)*	Indicates an error occurred either during a DMA transfer or programming of an FPGA(s). Once this bit is set, all controller operations are halted, and the controller must be reset to resume normal operation.
7	**Reserved**	N/A	This bit is not used for any purpose, but functionality can be added in the future.

Control2 (Bits 8-15)			
Bit	Function	Enable Value	Description
8-11	**FPGA[1-4]**	ON=1 *(R/W)*	These bits determine which FPGAs are intended for the DMA or programming requests.
12-15	**FPGA[1-4] Bank**	BANK 0 = 0 BANK 1 = 1 *(R/W)*	These bits determines the bank of configuration memory that will be used during a DMA transfer or while programming. Each bit corresponds to an FPGA. A value of '0' refers to bank 0, and a value of '1' refers to bank 1. (NOTE: With devices larger than Xilinx XC4013, there is actually only one bank of configuration memory since more than 32KB is required for one bitstream configuration.)

Table 2.1: *A description of each bit in the two control registers.*

CR2 Port Address 0x1

	FPGA4 Bank	FPGA3 Bank	FPGA2 Bank	FPGA1 Bank	FPGA 4	FPGA 3	FPGA 2	FPGA 1
Bit:	15	14	13	12	11	10	9	8

CR1 Port Address 0x0

	Reserved	Error (Read Only)	Clock On	Bitstream	Program	DMA On	Interrupts Enabled	Reset
Bit:	7	6	5	4	3	2	1	0

Figure 2.5: *The RACE controller's* **Control Register**.

count registers are both 17 bits wide, though it can only be accessed 8 bits at a time. Consequently, each register is divided into three 8-bit registers with the most significant register (Address3 and Count3) having only one writable bit (bit 17 of the address and count registers). The other bits of Address3 and Count3 are fixed to 0x0.

User-Definable Registers and Control

The RACE controller can be optimized to work with specific applications that a user may want to implement on the system. For example, suppose an application is divided into four individual bitstream configurations (or partitions) with each configuration being implemented on one of the four FPGAs on the RACE board. Partition 1 may use all of its data and need to have the next section of data transfered from the host's memory onto the board. Connected to the controller are several uncommitted signals that are attached to each FPGA. Inside the user-definable portion of the controller, registers could store status information from each partition, which the host's software could read through slave accesses to the controller. The partition can notify the controller that it requires the next section of memory, and then the software, upon reading this status information from the controller, could initiate the next DMA transfer of memory to RACE. Similarly, the user-definable portion of the controller can also be used to synchronize the different partitions since they may execute at different rates.

DMA Transfers

DMA transfers, or **D**irect **M**emory **A**ccesses, are the only way that data can be transfered between the RACE memory and the host (i.e., not considering the controller's registers accessible through slave accesses). In order to perform DMA transfers, a DMA kernel device driver has to be install that actually programs both the LSI L64853A and the RACE controller. To initiate a DMA transfer, first the RACE controller's Control, Address, and Count registers are loaded with the appropriate values to begin the DMA transfer. Next, the L64853A's CSR is programmed with the proper enable values, the Address register is loaded with the virtual address of the memory to be transfered, and the Byte Count register is loaded with the size of the data transfer. Once the RACE controller is programmed, it initiates the DMA transfer with the L64853A by

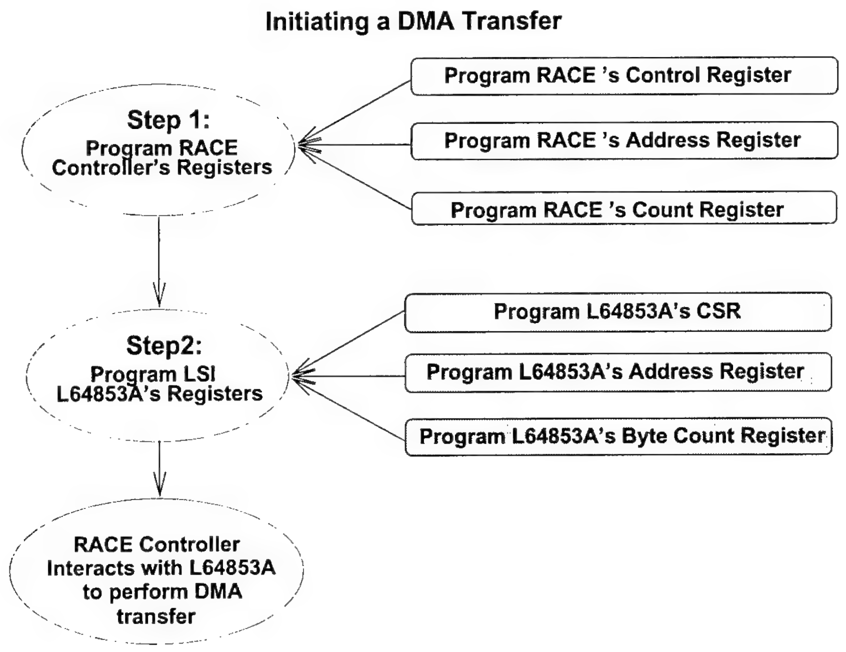
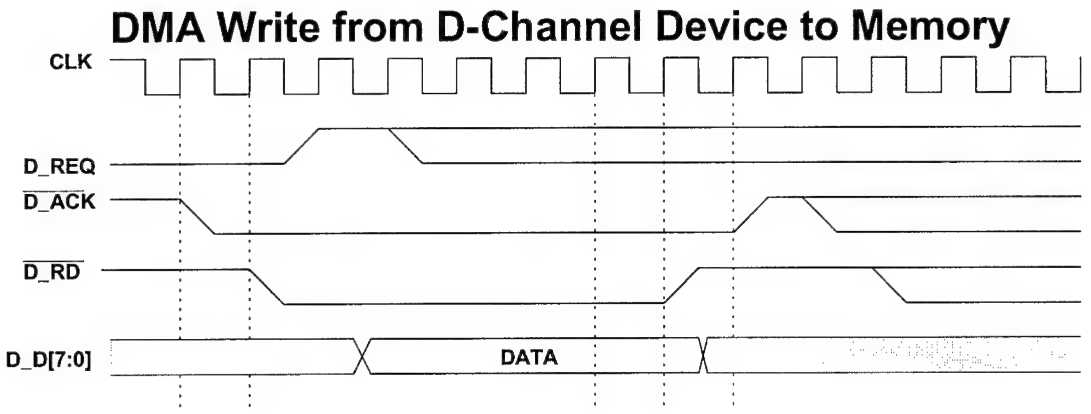

Initiating a DMA Transfer
Step 1:
Program RACE Controller's Registers
Program RACE 's Control Register
Program RACE 's Address Register
Program RACE 's Count Register
Step2:
Program LSI L64853A's Registers
Program L64853A's CSR
Program L64853A's Address Register
Program L64853A's Byte Count Register
RACE Controller Interacts with L64853A to perform DMA transfer

Figure 2.6: *Steps to initiating a DMA transfer.*

asserting the D_REQ signal and the transfer begins. Each byte of data is accompanied by a $\overline{D\_ACK}$ and $\overline{D\_RD}/\overline{D\_WR}$ (depending on the direction of the data transfer) from the L64853A. When the DMA transfer is complete, the RACE controller releases the D_REQ signal and the transfer stops. Figure 2.6 shows the steps required to initiate a DMA transfer, and Figure 2.7 illustrates the signals involved in a DMA transfer.

Configuring the RACE Controller

The RACE controller can be programmed in three different ways: 1) slave serial via an external XChecker cable (provided from Xilinx), 2) master serial by means of an on-board SPROM, or 3) asynchronous peripheral by means of the on-board PAL. In slave serial mode, an XChecker cable programs the controller one bit at a time through the workstation's serial port. This mode is useful for rapid prototyping of new controller designs. If the controller's configuration will not change, then the SPROM (serial PROM) can be used to store the controller's static configuration. However, the most convenient way of dynamically configuring the controller is through asynchronous peripheral mode. The PAL has its register written to through a slave access, which initiates the reprogramming of the controller. The PAL then acts as the interface between the L64853A as the bitstream configuration is written one byte at a time to the controller through slave writes. Once the controller is configured, the PAL is no longer needed as an interface to the L64853A. The configuration mode of the controller is determined by setting the four dip switches adjacent to the controller. Each mode signal, M0-M2, can be turned on or off depending on the desired mode (see [24] for the available configuration modes and mode signals settings), and the **ENABLE_SPROM** must be turned on if the SPROM is to be used for master serial mode.



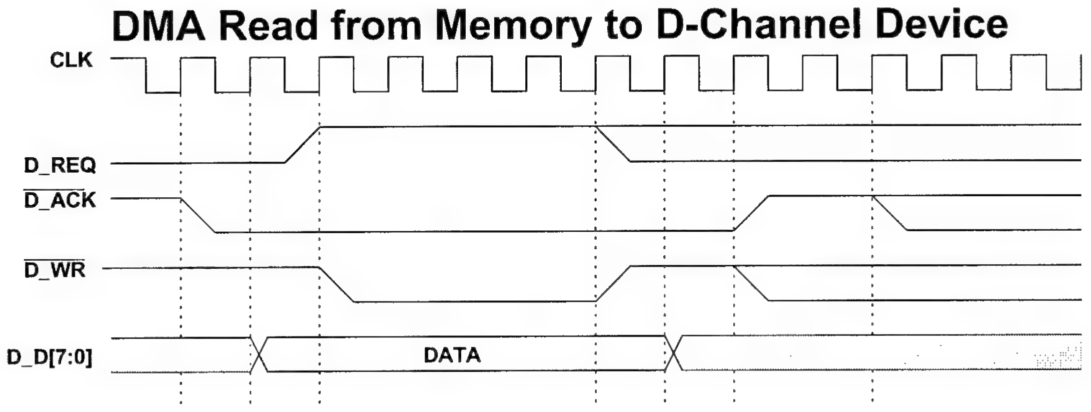


Figure 2.7: *L64853A D-Channel signals during a DMA transfer.*

2.4.2 RACE Memory

The memory used on the RACE board consists of 15ns access time, 32x8KB SRAM. Each FPGA has two types of memory local to itself: **configuration memory** (or also called **bitstream memory**) and **data memory**. The configuration memory is used for one purpose only—to store bitstream configurations for the local FPGA. Sixty-four kilobytes of SRAM is specifically set aside for each FPGA for storing bitsteams. Since the RACE board uses Xilinx XC4013 FPGAs, approximately 30KB is required to store one bitstream configuration. Consequently, the 64KB can be divided into two banks of 32KB, allowing the storage of two bitstream configurations. (Larger FPGAs like XC4020 and XC4028 require around 50KB for a bitstream; therefore, only one bank of memory would exist using these FPGAs). Storing bitstreams on the board eliminates the overhead incurred by the DMA transfer of the bitstream to the board by allowing the FPGAs to be programmed immediately once the current implemented function is finished. Consequently, different applications can be queued-up in the configuration memory while waiting to be implemented in the FPGA, which further facilitates reconfigurable computing.

The data memory local to each FPGA can be used by the functions as local data storage. If a function needs more SRAM than what can be provided on the FPGA itself, then it can utilize the 128KB of memory hardwired locally to the FPGA. Each FPGA has identical pinouts to its local memory so functions can be implemented on any FPGA on the board and still work. Since DMA transfers can be occurring to the configuration memory while an application is executing on the FPGA, the controller performs a trivial memory arbitration. The application has access to its local memory as long as the memory grant signal from the controller is active low ($\overline{FPGA\_MG}$).

Functions implemented in the FPGAs can read and write to the data memory by asserting the active low signals $\overline{FPGA\_RD}$ and $\overline{FPGA\_WR}$ that are hardwired from each FPGA to its local memory's output enable and write enable signals, respectively. The configuration memory's output enable and write enable are directly connected to the $\overline{D\_RD}$ and $\overline{D\_WR}$ signals (which come from the L64853A), respectively. Ultimately, the controller determines how the memory functions on the RACE board. By asserting the appropriate $\overline{FPGA\_RAM}$ and $\overline{FPGA\_BRAM}$ signals, it can turn on any and every memory bank at any time. Therefore, for applications like MISD (*multiple instructions, single data*) applications, all the data memory could be activated simultaneously during a DMA write so that each FPGA receives the same data. Likewise, for SIMD (*single instruction, multiple data*) applications, the same bitstream configurations could be written to the configuration memory local to each FPGA. When each FPGA is programmed, they could all be implementing the same function, but with different data. By modifying the RACE controller, the user has a wide flexibility in how the RACE memory operates.

2.4.3 RACE FPGAs

The RACE FPGAs are the most significant component of the RACE system since they make reconfigurable computing possible. Four Xilinx XC4013MQ208-5 FPGAs are available on the RACE board. Each FPGA has identical pinouts, which prevents an application from being constrained to a particular FPGA on the board. Likewise, having identical pinouts allows applications stored in the hardware library to be implemented on whatever resources are available even if other applications are executing simultaneously. Since large applications may be partitioned across multiple FPGAs, a fully connected K_4 interconnection exists between the four FPGAs. The bus width for each edge of the K_4 interconnect is 32-bits. Each FPGA can be thought of as having three neighbors—one to the east, south, and diagonal (southeast). The FPGAs are rotated on the board such that a particular set of pins on each FPGA is connected to its neighbor to the east, south, and diagonal (see Figure 2.8). For example, let the east, diagonal, and south interconnects be called A, B, and C, respectively. Suppose pins 1-32 are assigned to A, pins 33-65 are assigned to B, and pins 66-98 are assigned to C. An application is partitioned across the four FPGAs and must communicate between each partition; therefore, the partitioner needs to determine which pins to use for the inter-FPGA communication. It determines that the partition on FPGA1 needs to send a signal to its east neighbor using signal A1. The FPGA east of FPGA1 would be FPGA2 on the RACE board. Since FPGA2 is rotated 90 degrees from FPGA1, A1 connects into C1 of FPGA2 (i.e., FPGA1 is "south" of FPGA2). Therefore, the partitioner knows that the signal appearing on C1 is coming from FPGA1's A1 signal. In the case of diagonal neighbors, a signal appearing on B1 from FPGA1 would also appear on B1 of FPGA3. Figure 2.8 illustrates the K_4 network between FPGAs.

2.5 RACE Prototype

A prototype of RACE was made before the full PCB version was created. The prototype was used to not only to experiment with reconfigurable computing, but to gain some valuable experience in implementing a reconfigurable hardware platform. The prototype is a much smaller wire-wrap version of the full RACE

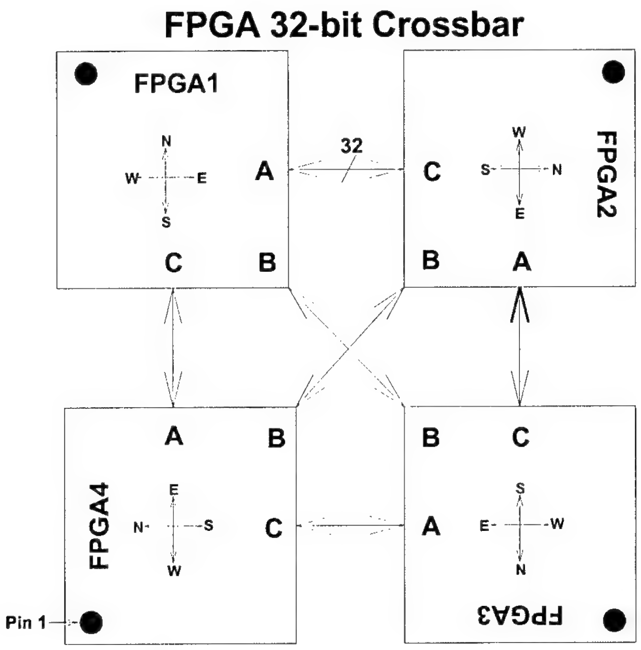


Figure 2.8: *The K4 interconnections between FPGAs on the RACE board.*

configuration and can only implement one function at a time. The wire-wrapping was performed on a large MUPAC VME-size wire-wrapping board.

Since the Xilinx XC4013 FPGA comes only in a 208-pin package or higher, smaller FPGAs were used in the prototype. A Xilinx XC4005-5 in an 84-pin PLCC package was used for the controller, and a XC4010-5 (84-pin) was used for implementing functions. (Only one FPGA was used for functions because of the difficulty of hand wire-wrapping not only four FPGAs, but all the control signals and SRAM.) The controller is programmable only through an external XChecker cable (from Xilinx) that downloads its bitstream configuration from the workstation's serial port. Furthermore, the controller's logic is greatly reduced since only one FPGA needs to be controlled. The control and flag registers consist of only 8 bits, and the address and count registers consist of 16-bits. Since the prototype controller has fewer registers, only three physical address signals from the SBus needed sent through the ribbon cable to the controller. Slightly more control logic is consumed for decoding the appropriate memory locations since a 74ALS139 was not used on the prototype. Clocking the controller FPGA at 25MHz was very difficult since only slight configuration modifications resulted in too long of path delays on the critical paths. In fact, the final working version of the controller was determined by XDELAY to only clock at 19MHz; however, it clocked fine at 25MHz. The most timing critical part of the controller is during DMA transfers when it needs to handshake with the L64853A and generate the appropriate addresses on the address bus. Such timing concerns led to the decision of using the Xilinx 4000E-series FPGA for the controller on the full configuration.

The XC4010 on the prototype board had fewer signals connecting it to the controller than the FPGAs on the full RACE configuration. On the full configuration, the controller has eight dedicated interconnects that the FPGAs can use to communicate to the controller. However, on the prototype, only a status and error signal communicate to the controller what is happening on the FPGA. Moreover, the XC4010 is programmable in only two modes—master parallel up and down. A programmable external clock can be used, but it only

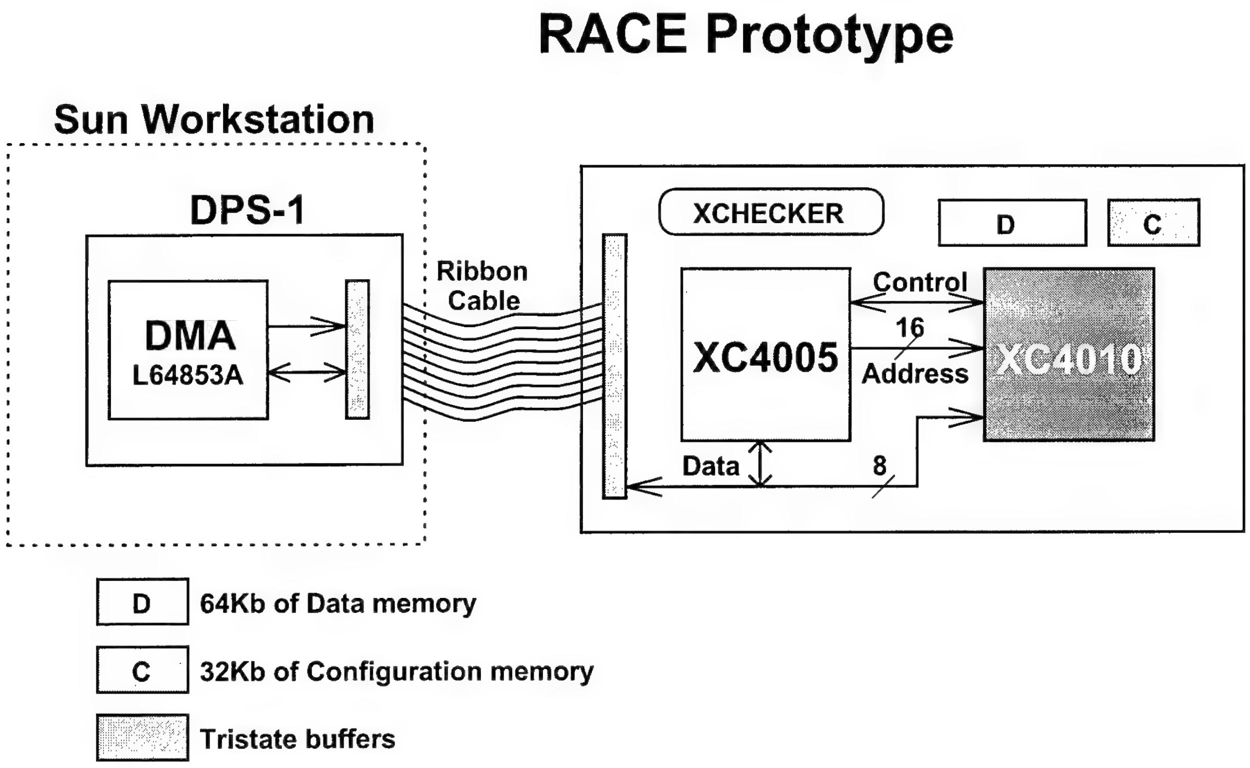


Figure 2.9: *The RACE prototype. The prototype is a wire-wrapped version that consists of fewer memory, a smaller controller, and only one FPGA.*

clocks at 24MHz, which makes dividing the global 25MHz inside the XC4010 just as easy to use. Figure 2.9 illustrates the prototype's configuration, and Figure 2.10 shows an actual picture of the wire-wrapped prototype.

2.6 NEBULA Architecture

NEBULA is a partially reconfigurable PCI coprocessor. As described earlier it caters to a class of applications which are time consuming in execution on a fixed processor architecture. Floating point operations in general are not very good for execution on FPGAs. Digital signal processing algorithms go very well with this architecture since the reconfigurable logic could be used to pipeline and parallelize the operations required by any algorithm. The class of applications which would enable a designer to parallelize the operations and pipeline them in reconfigurable logic is the one that is suitable for this coprocessor board. In addition to the partially reconfigurable units, NEBULA has a huge fast memory bank which could be used by the reconfigurable logic.

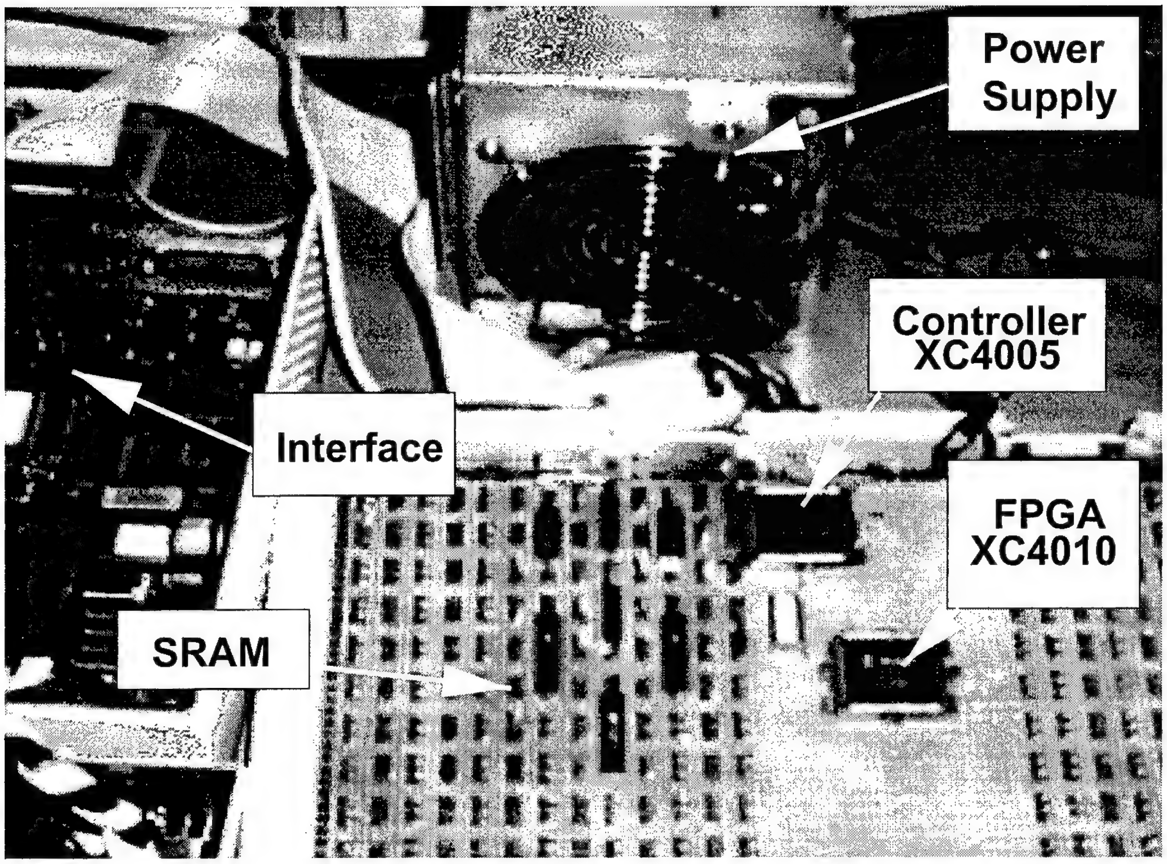


Figure 2.10: *Picture of the wire-wrap prototype of RACE. The wire-wrap board consists of two FPGAs, tristates, and SRAM connected to the Sun workstation through the DAWN DPS-1 DMA interface.*

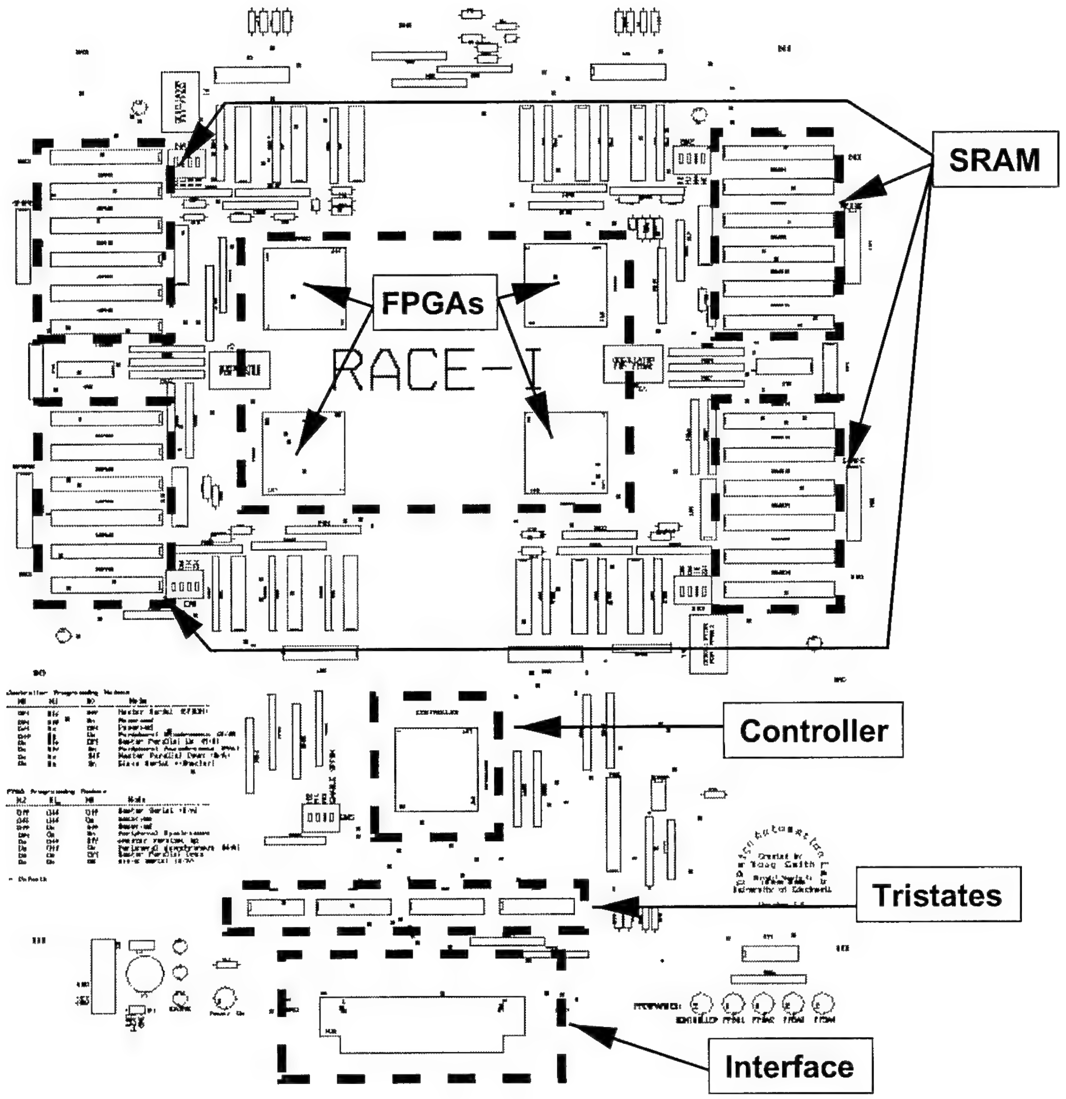


Figure 2.11: *Outline of the PCB layout of the RACE board.*

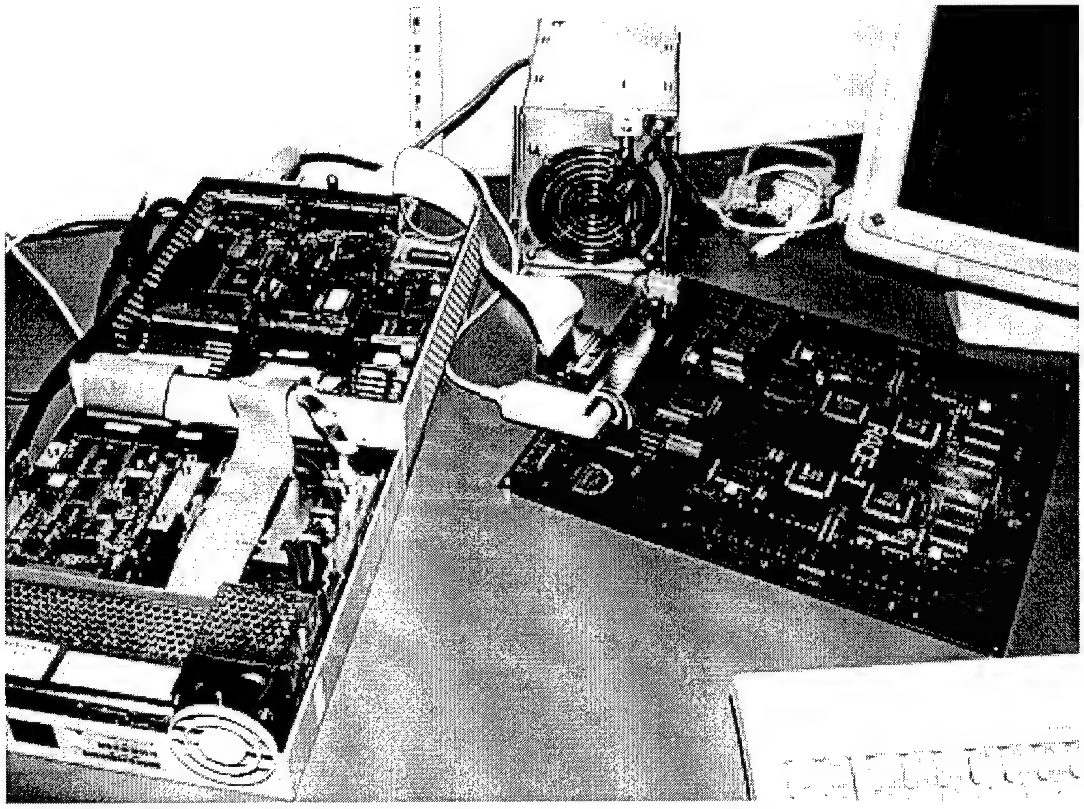


Figure 2.12: *Picture of the completed RACE PCB.*

2.6.1 NEBULA: Goals

The main goal of the NEBULA architecture was to enable a class of applications to be executed very quickly compared to the execution on a fixed general purpose architecture processor. Quite a large number of reconfigurable platforms have been designed to take advantage of the higher speed of execution on the hardware. Our next goal was to provide improvisations over the other existing reconfigurable platforms. Most of the existing reconfigurable platforms are static in nature. The goal of this project was to go one step further and use partially reconfigurable logic which would enable rapid dynamic reconfiguration. An additional goal was to have a huge local memory so that the coprocessor could make use of the entire range of memory for memory hogging applications which is very common in Digital Signal Processing algorithms.

2.6.2 NEBULA: Environment

The NEBULA environment consists of a custom high performance adaptive computing environment capable of fine, medium and coarse grain reconfiguration. The NEBULA card is a dynamically reconfigurable PCI coprocessor target board. This is housed in a Dell 300 MHz PC on one of the PCI slots. The PCI interface provides a 33MHz clock and 32-bit multiplexed address-data bus. The PCI interface block is a configurable unit on NEBULA. On boot-up, a serial PROM loads the PCI interface onto the controller FPGA of NEBULA. The system software detects the PCI card by reading some of the PCI configuration

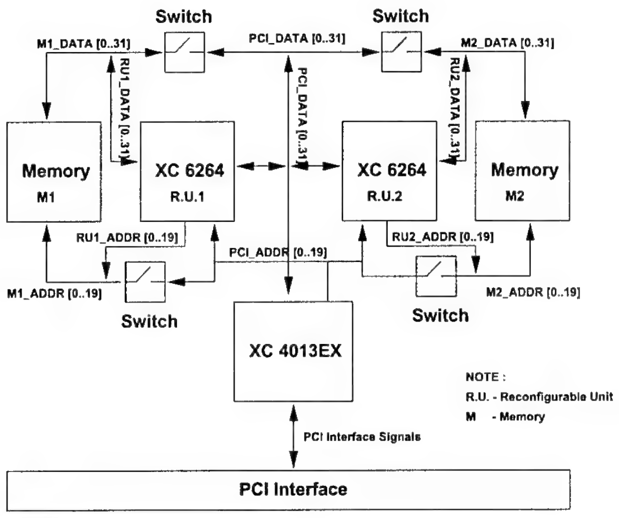


Figure 2.13: *NEBULA : Architecture*

registers. Once the card has been detected and information about the card has been registered by the system, further accesses can be made by using the device drivers written for the card. A mezzanine connectivity is also provided on NEBULA for future expandibility.

The initial phase of using this environment is to identify parts of an application which could be mapped onto the coprocessor. The configuration streams required for the execution of that part of the code with an envelope of calls to device driver routines is then developed. Then the whole application is executed as a software/hardware co-execution.

2.6.3 NEBULA: Features

NEBULA is a coprocessor board which aids the host processor by executing applications mapped onto it in parallel with the processor software execution. Some of the features of NEBULA :

- 2 Xilinx XC6264 family chips as partially reconfigurable units
- 2 MB Fast SRAM
- 1 Xilinx XC4020E-01 for PCI interface
- 2-100 MHz Programmable Oscillator
- Mezzanine Connectivity for expandibility

2.6.4 NEBULA : Architecture

The NEBULA environment as discussed earlier consists of a host machine (Pentium 300 MHz Dell machine) with the NEBULA card occupying one of the PCI slots in the machine. The NEBULA architecture is depicted in Figure 2.13. It comprises of two reconfigurable units with local memory directly accessible by the host CPU through the PCI interface. The configuration data for the reconfigurable units and the data for the applications are transferred from the host memory using the PCI interface.

2.6.5 Local Bus Standard - PCI

One of the goals of the NEBULA card was to reduce the reconfiguration overheads and the data communication overheads between the host processor and the coprocessor. The Peripheral Component Interconnect Bus (PCI) provides a 33MHz clocking frequency and a maximum data transfer rate of 132 MBps. This high data transfer rate is possible because of the high clocking rate and a wide data bus width of 32 bits. All the other local bus standards are inferior in comparison with the PCI, taking into consideration the data transfer rate. Also, the PCI standard allows burst transfers of data which is a requirement for high speed data transfers from the host main memory to the coprocessor local memory. For further details on the PCI standard, refer to the chapter on PCI Bus Standard.

2.6.6 Main Controller

The Main Controller comprises of circuitry for the PCI interface and also for controlling the events on the board. The Controller logic is configurable onto a Xilinx XC4000 family FPGA. The Main controller consists of two main logic blocks

- PCI Logic Core
- NEBULA Control Core

The Xilinx LogiCORE [26] for the PCI interface is used for the PCI interface. Additional logic is used for controlling various events on the coprocessor. More details on the Controller are given in further sections.

The *PCI logic core* consists of a Parity generator/Checker, an Initiator State machine, a Target State machine and implementation of some configuration registers.

The NEBULA control core consists of a host of user registers used to control access mechanisms, interrupt controls and on-board clock programming. The control generation for accessing on-board memory and the reconfigurable units is a part of this block. This block also has the interrupt generation unit which generates an interrupt to the host processor depending on certain events that take place on the coprocessor.

2.6.7 Memory organization and Reconfigurable Units(RU)

One of the main issues in the design of the reconfigurable architecture is the provision of local memory or global memory. The provision of global memory would involve implementing a huge chunk of memory which can be accessed by all the reconfigurable units and the system CPU. With the implementation of global memory comes the problem of handling memory management and preventing memory access conflicts. Provision of local memory for the RUs on the other hand makes them independent units which can perform parallel computations without conflicting access of memory. NEBULA provides a huge local memory of 1Mbytes for each RU. The memory can be accessed in byte, word or long word formats. Thus highly data intensive applications can be implemented on this architecture.

NEBULA provides partially and dynamically reconfigurable units using Xilinx XC6200 family FPGAs. The XC6000 family FPGAs have a very small reconfiguration time compared to the Xilinx XC4000 family FPGAs. On top of this these FPGAs are partially reconfigurable. The reconfiguration overhead is dependent upon the size of the design unlike that in XC4000 family FPGAs where the reconfiguration overhead is a constant. Small changes in design could be easily made in a matter of a few hundreds of nano seconds. This ability opens up a new approach towards designing circuits. An architectural issue was to decide on the number of RUs to provide on the coprocessor board. The space limitation of the long PCI card was a

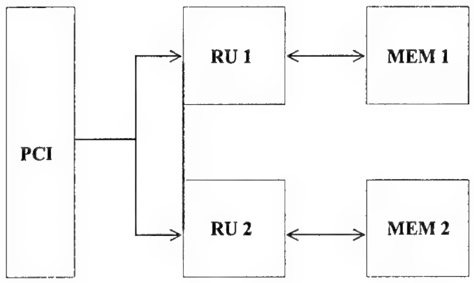


Figure 2.14: *Access Mode 1*

major factor in deciding on the number of RUs to be two. Moreover, the lesser the number of RUs, the simpler is the task of partitioning a huge design onto the RUs. Since there are only two RUs, partitioning is just a bipartitioning problem which is well understood by the CAD community. The two RUs used in NEBULA is equivalent to somewhere between 128k to 200k gates. A very large interconnect bus of seventy lines connects the two RUs. Very large applications could be partitioned and mapped onto the two RUs.

2.6.8 Access Mechanisms

The interconnectivity between the RUs, memory and the PCI interface is shown in the Figure 2.13. The RUs and the local memory of NEBULA can be accessed by the host processor through the PCI interface. The local memory could also be accessed by the RUs. However both the processor and the RUs should not access the memory simultaneously. In order to avoid these conflicts, certain hardware precautions are taken to prevent this possibility. The various legal access mechanisms are described in this section.

Access Mode 1

Figure 2.14 shows the Access Mode 1. The host processor accesses both the RUs (RU1 and RU2) simultaneously. During this period, the RUs can access their local memory. This mode is useful when a particular functional unit has to be configured on both the RUs. Instead of individually configuring them, it would be very convenient to configure them simultaneously. Also, this mode is useful when the host processor wants to dump some data onto the state registers on the RUs. While the host processor is configuring the RUs or dumping data onto the RUs, the RUs themselves could be accessing their local memory. This allows an RU to continue with its execution while the host processor modifies another part of the RUs without interfering with the execution of the RUs.

Note that the simultaneous access to the RUs is allowed only for the "write transaction". This is not allowed for simultaneous read transactions since different data may be driven from the two RUs resulting in damage to the external buffers.

Access Mode 2

Figure 2.15 shows the Access Mode 2 configuration. The host processor accesses the local memory of both the reconfigurable units simultaneously. This is useful in instances where both the RUs are operating on the

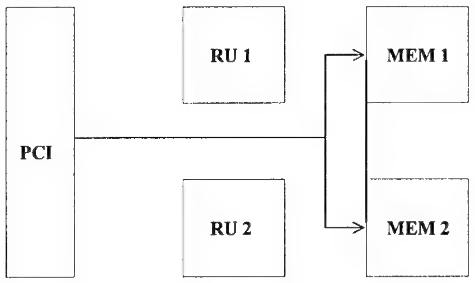


Figure 2.15: *Access Mode 2*

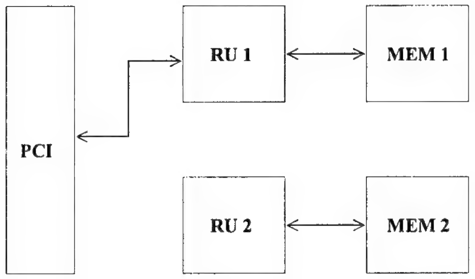


Figure 2.16: *Access Mode 3*

same data set and the data needs to be dumped from the system memory onto the local memory of the board. As previously noted, only simultaneous write transactions are allowed in this mode too.

Access Mode 3

Figure 2.16 shows the configuration for Access Mode 3. It is very similar to Access Mode 1. However, simultaneous access of RUs is not made. The host processor accesses one of the RUs and the RUs individually can access their local memory.

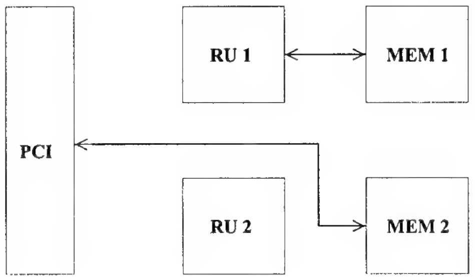


Figure 2.17: *Access Mode 4*

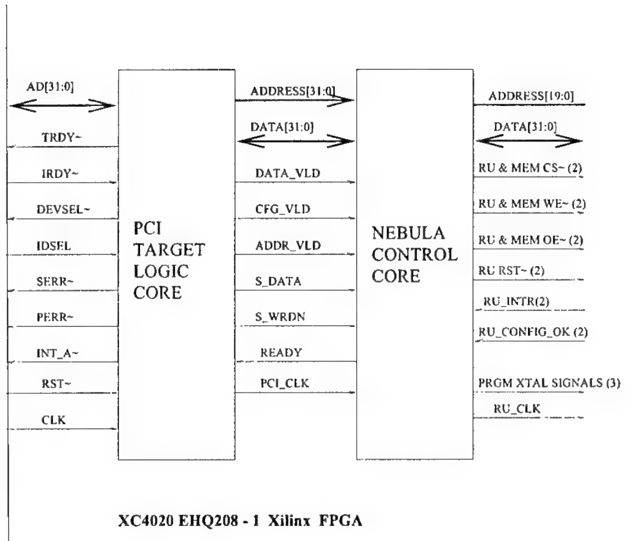


Figure 2.18: *Main Controller*

Access Mode 4

Figure 2.17 shows the configuration for Access Mode 4. In this mode, the host processor accesses the local memory of one RU while the other RU optionally accesses its own local memory. This mode is useful since even during the execution period of the RUs, one of the local memory units could be filled by the processor making data ready for the other RU.

2.6.9 NEBULA : Circuit Description

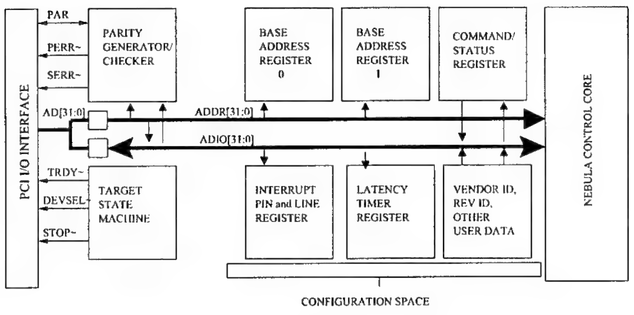
This section describes the actual implementation of the NEBULA card. The main blocks of the NEBULA are the following.

- Main Controller
- Memory
- Reconfigurable Units
- Buffering block
- Programmable Clock Oscillator
- Mezzanine Connectivity

Each of these blocks will be discussed in detail.

2.6.10 Main Controller

The Main Controller (refer Figure 2.18) resides on a reconfigurable Xilinx XC4000 family FPGA. The particular part used on the NEBULA is a XC4020EHQ240-1. The speed grade used is -1 which is the



LogiCORE PCI Interface Block Diagram

Figure 2.19: *PCI Logic Core*

fastest speed grade for the part. This speed grade of the FPGA is used mainly because the PCI interface design has many critical paths and the timing for these paths are met only by using a very fast FPGA. There are two main blocks in the Main Controller. One is the PCI interface logic core and the other is the Nebula control core. A PCI Target LogiCORE from Xilinx has been used for handling the PCI interface of NEBULA. Additional logic is implemented to control the various aspects of the NEBULA card.

2.6.11 PCI Target LogiCORE

The LogiCORE PCI interface is a PCI interface building block created for XC4000E FPGA family. The detailed block diagram shown in Figure 2.19 forms the core PCI interface design.

The LogiCORE supports a complete 32-bit PCI interface. It is PCI local bus compliant for Revision 2.1.

2.6.12 Nebula Control Core

Figure 2.20 shows the Nebula Control Core organization. Both the memory and reconfigurable units are accessible from the host processor through the PCI interface. Proper control mechanisms for accessing these units and control signal generation for these units is done in the Nebula Control Core. The individual blocks within the Nebula Control Core are described here.

Write Enable Generation

In order to access the user registers, memory and the RUs, control signals have to be generated using the signals that the *PCI logic core* provides. The *PCI logic core* block provides various PCI control signals which are to be used for the generation of "Write Enable" signals. Write Enable signals have to be generated for the user registers, memory and the RUs.

Clock Selection Block

This block receives three clocks as its inputs : 33 MHz PCI clock, 16 MHz crystal clock and the clock from the programmable clock generator (2MHz to 100 MHz). One of these clocks is selected depending on the User Register values for the clock selection bits and is given out as the global clock to the RUs. However, the 33MHz clock has to be used for configuration and state accesses of the RUs since the read and write

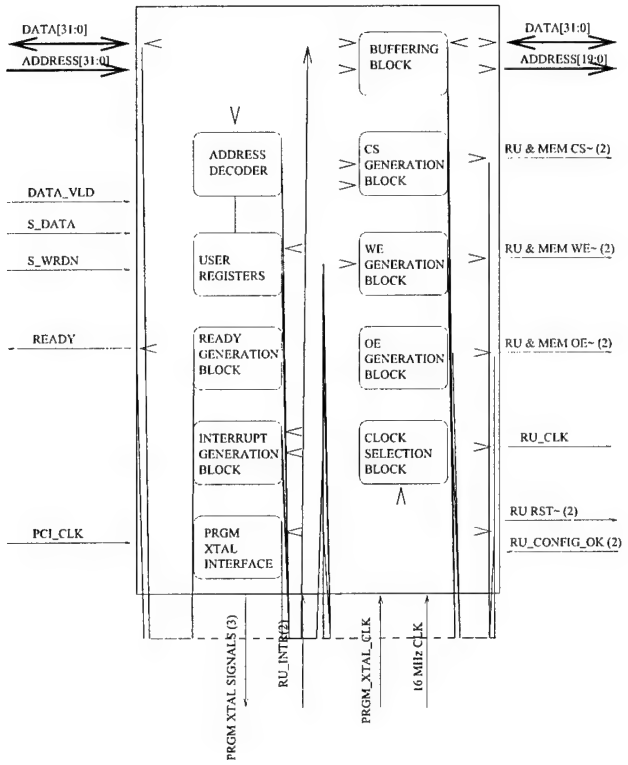


Figure 2.20: *Nebula Control Core*

transactions are synchronized to the PCI 33MHz clock. Once the RU is configured, any particular clock could be used for the user design on the RUs.

Buffering block

All the address and data lines required for the memory and RUs are buffered in this block. All inputs and outputs are clocked with the 33MHz PCI clock so that everything is synchronized to a particular reference clock. The control signals like chip select, write enable and output enable are also buffered and driven out of the Main Controller.

2.6.13 Memory

On-board memory of 2 MB fast SRAM is used in NEBULA. This corresponds to 1MB local memory for each RU. The part used for memory is an integrated chip from *Samsung*. The *Samsung* memory part used on NEBULA is KM6164002-20. It is a 256K x 16 Bit High Speed CMOS Static RAM. It works on a power supply of 5V. It provides TTL compatible inputs and outputs, fully static operation, three state outputs, center power/ground pin configuration and data byte control for accessing higher and lower bytes. The part provides an access time of 20ns for the read operations. It is available as a 400 mil 44pin plastic SOJ package. Two such memory parts are used for each RU. The control signals for the memory can be generated by either the Main Controller or the RUs.

Property	XC6264 details
Typical Gate Count Range	64000-100000
Number of cells	16384
Number of registers	16384
Number of IOBs	512
Cell Rows x Columns	128 x 128

Table 2.2: *XC6264 data specifications*

2.6.14 Reconfigurable Units

The reconfigurable units used in NEBULA are the Xilinx XC6200 family FPGAs. These FPGAs are partially and dynamically reconfigurable. Two XC6264 parts are being used on this coprocessor. The details of the XC6200 family of FPGAs is described in brief in this section.

2.6.15 The Xilinx XC6200 Family of FPGAs

The XC6200 family FPGA [2] is a high performance Sea-Of-Gates FPGA. It is a fine grain architecture with abundant registers, gates and routing resources. It is an Advanced Processor compatible architecture. It provides a Xilinx FastMAP processor interface which enables direct processor read/write access to all internal registers in user design with no logic overhead. It also provides a programmable data bus width and is easily interfaced with most microcontrollers and microprocessors. It allows high speed reconfiguration via the parallel CPU interface and is ideal for custom computing applications. It has a flexible interconnect architecture and provides a low-delay FastLANE hierarchical routing scheme. It also provides flexible pin configuration like any other FPGA. The development on the XC6200 FPGAs is supported by CAD tools like Viewlogic, Synopsys and XACT step series 6000 backend tools.

2.6.16 RU organization on NEBULA

Partially and dynamically reconfigurable XC6264 FPGA from Xilinx is used as the reconfigurable unit on NEBULA. The details of this particular part are shown in Table 2.2.

Two XC6264 FPGAs are used as reconfigurable units on NEBULA. The address and data bus from the Main Controller are used to access the RUs using the FastMAP interface of the XC6264. Control signals like *Chip Select* and *Write Enable* are generated in the Main Controller. The two RUs are interconnected with seventy interconnecting signals. Hence if a huge design were to be partitioned and mapped onto the two RUs then the cutset between the partitions can be as high as seventy. The global clock to the RUs is generated by the Main controller. The PCI 33MHz clock is used for configuring the RUs and for accessing registers within the RUs. However, the user design could make use of the PCI 33MHz clock 16MHz or the programmable clock as its global clock for execution.

The RUs are connected to the local memory modules also. A set of pins are allocated for the address and data lines towards the memory modules. The user design on the RUs should provide the address and data on the assigned pins of the XC6264 to access the local memory. However the user of the RUs should ensure that the user design does not access the memory when the Main controller is accessing the same memory unit.

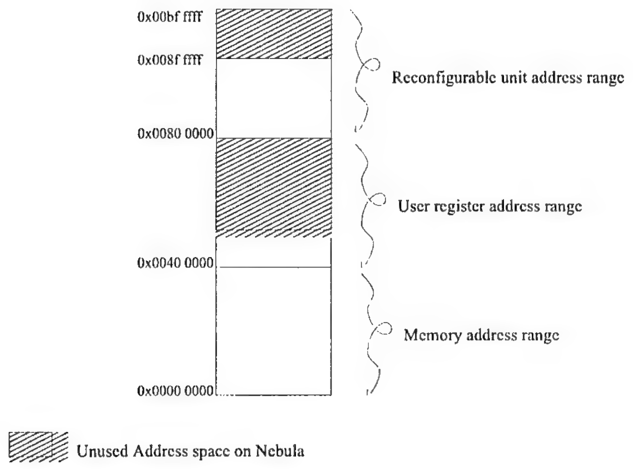

0x00bf ffff
0x008f ffff
0x0080 0000
0x0040 0000
0x0000 0000
Reconfigurable unit address range
User register address range
Memory address range
Unused Address space on Nebula

Figure 2.21: *Address Ranges for Memory, user registers and reconfigurable units*

2.6.17 Mezzanine Connectivity

Three 64 pin IEEE 1386 mezzanine connectors are used on the board. These are used to enable a child card to be connected to the Nebula board. The part number that is used for the mezzanine connectors is 120527-1 from AMP. Twenty address lines, thirty two data lines and a few control signals from the Main Controller (PCI interface FPGA) are brought to the mezzanine connectors. Twenty four signals from the interconnect network between the two reconfigurable units is also brought to the connectors. Eight unused lines from the Main Controller FPGA and four unused lines from each of the reconfigurable units are also connected to the mezzanine connectors.

2.6.18 Nebula Programming Considerations

The Main Controller controls the generation of control signals for the reconfiguration unit and memory on the board. It also has a set of user registers which can be programmed for controlling the generation of these control signals. The three main address spaces on the Nebula board are:

- Memory
- User Registers
- Reconfigurable units

The User registers are both memory mapped and I/O mapped and hence can be accessed as either memory locations or I/O locations. Address bits AD23 and AD22 are used to decode the particular address space for each of these units. Figure 2.21 shows the address range for these units.

The address space allocated for memory is four times that of the size of a single unit of memory. Each memory unit is 1MB and the space allocated for this is 4 MB in the address space. This is due to the fact that address lines AD[21:2] are used for addressing memory. The reason for using this set of address lines is addressed in the next paragraph. By using control bits in the user registers, the user can decide which memory unit to access. It is possible to access both the memory units simultaneously for write operations.

The address space allocated for the reconfigurable units is also four times the address space required for a single reconfigurable unit. The XC6264 is addressed by eighteen address lines and hence requires a memory allocation of 256 KB. However 1MB is allocated in the address space since the address lines AD[19:2] are used to address the RUs. The reason for using these address lines is to enable the host processor to make single byte accesses to the RUs. The XC6264 gives out its data on the D[7:0] lines or the least significant eight data lines. The PCI protocol requires that the data should be presented on either the D[7:0] or D[15:8] or D[23:16] or the D[31:24] lines. i.e. for addresses 0x0, 0x4, 0x8, ... , a byte access is to be made on the D[7:0], for addresses 0x1, 0x5, 0x9, ..., byte access is to be made on D[15:8], for addresses 0x2, 0x4, 0x6, ... byte access is to be made on D[23:16] and for addresses 0x3, 0x7, 0xa, ..., byte access is to be made on D[31:24]. Since this is not possible for accesses to XC6264, the configuration has been designed so that only addresses AD[19:2] are used for addressing RUs. The lower two bits are always zero from the PCI side and hence will be considered as byte accesses on addresses 0x0,0x4,0x8... and the data is expected on D[7:0]. Since the same address lines are used for the memory units too, the address displacement has to be made for accesses to memory as well.

2.7 NEBULA PCB

In this section, a few details are described about the NEBULA printed circuit board. The size of the PCB was designed in accordance to the PCI specifications Revision 2.1s. The size conforms to the size of a standard length PCI expansion card and provides 49 square inches of real estate. Also the gold fingers for the PCI card was chosen from a standard library. The tools used to capture the schematics and layout and routing of the PCB were the Orcad Design Capture and Layout tools.

The Nebula design was made using surface mount components in order to maximize the area utility of the PCB. The buffers, FPGAs, RUs, resistors, memory and the mezzanine connectors are surface mount components which are mounted on the top of the board. The capacitors are also surface mount components and are mounted on the bottom of the board. The only through-hole components used in the design are the LEDs, PROM and the oscillator. A lot of test points are also made available for the GND, VCC and many control signals. All the data lines and the address lines from the Main Controller have been brought out to test points which helped us in debugging the board. Two LEDs are provided to indicate that the power is present and that the Main Controller FPGA is configured.

The thirty two width data bus and twenty line width address bus had to be routed to the memory and the reconfigurable units. These lines and the interconnection between the RUs made NEBULA a very dense board. Consequently, a six-layered, FR4-type material board was chosen for the implementation. The six-layered board had the following layer stacking: 1) top, 2) first inner layer, 3) power, 4) ground, 5) second inner layer, and 6) bottom. The top and bottom layers were routed vertically, and the two inner layers were routed horizontally. Initially all the power signals i.e. the 5V, GND, 3V and 12V signals were routed. Thick routes were used for the 3V and 12V lines. The power pins of the SMD components were fanned out. All the PCI signals from the gold fingers to the Main Controller FPGA were routed initially to satisfy the condition that all the PCI interface signals should be less than 1.5 inches in length. Then the address and data lines from the Main Controller to the buffers and the RUs were routed. Then the other signals were routed by giving priority to critical signals.

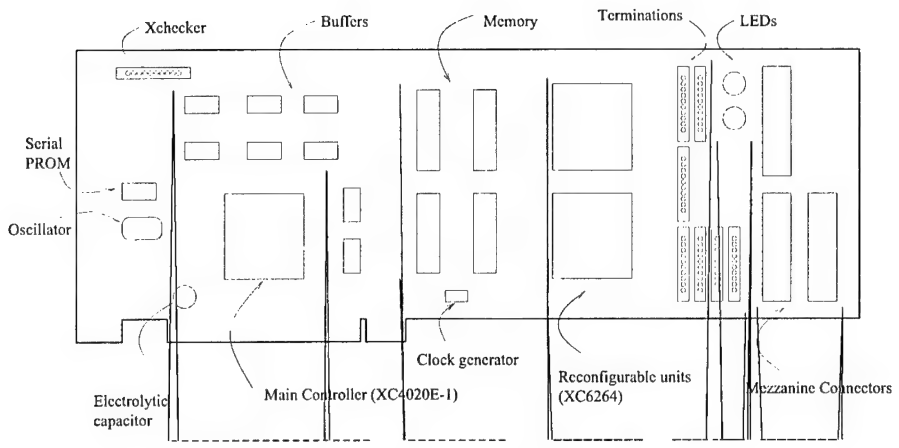


Figure 2.22: *NEBULA Layout*

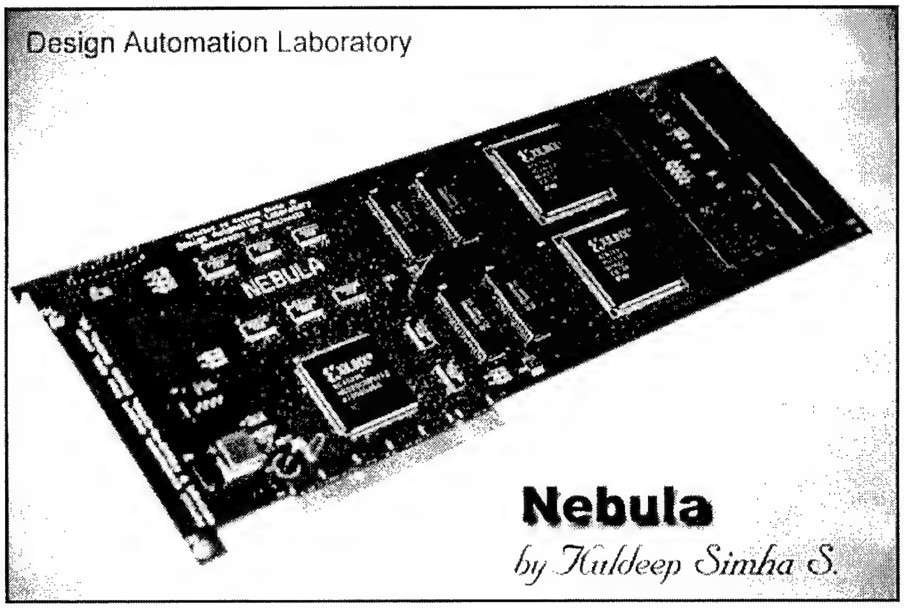


Figure 2.23: *NEBULA Card*

Chapter 3

Software

3.1 Issues in Programming Reconfigurable Computers

The programming methodologies of reconfigurable systems differ from the way normal computers are programmed. In normal computers the whole program executes on the general purpose microprocessor. But on a reconfigurable computer parts of the program are modeled as hardware on the reconfigurable hardware. The rest of the program executes as software on the general purpose microprocessor. The results of the hardware part are communicated to the main program executing on the microprocessor. The general idea is given in Figure 3.1. The main difference in programming reconfigurable computers as compared to programming on a general purpose computer is that *hardware equivalents* of software programs need to be generated so that they can be downloaded on to the FPGAs for execution. This usually calls for knowledge of CAD tools and hardware design, which the general programming community does not possess. Decision also needs to be made by the programmer on partitioning a program into hardware and software portions.

3.1.1 Partitioning programs between hardware and software

Designing systems containing both hardware and software components is not a new problem. A key point to address is the evaluation of execution time, since the partition to be produced should minimize it. Some criteria are needed to split the system under development into hardware and software parts. The usual constraints are performance, hardware area, and cost criteria. Once the hardware portions are identified, circuits need to be synthesized for those portions. A software and hardware evaluation should then made in order to check if the performance and area requirements are reached. If not, the partition should be revised. This is a sort of *repetitive cycle* which takes time and requires a lot of manual interaction.

When designing the hardware portions to execute on the the reconfigurable hardware, care should be taken only to move the computationally intensive portions of the algorithm onto hardware. Taking non-compute intensive portions of the code on to hardware might actually slow down the overall performance due to the overheads associated with programming the FPGA device.

The figure below Figure 3.2, gives a possible design flow involving a hardware/software partitioning of a program, so that it can be implemented on a reconfigurable computer. The source program is profiled to identify the computationally intensive portions of the algorithm. Profiling can be done using specific tools (*gprof* on unix) or manually by the user. Once the profiling of the code is done and the compute intensive portions of the program identified, hardware for the compute intensive portions can be generated. Note that the hardware generated should contain the necessary control circuitry to help it interact with the host

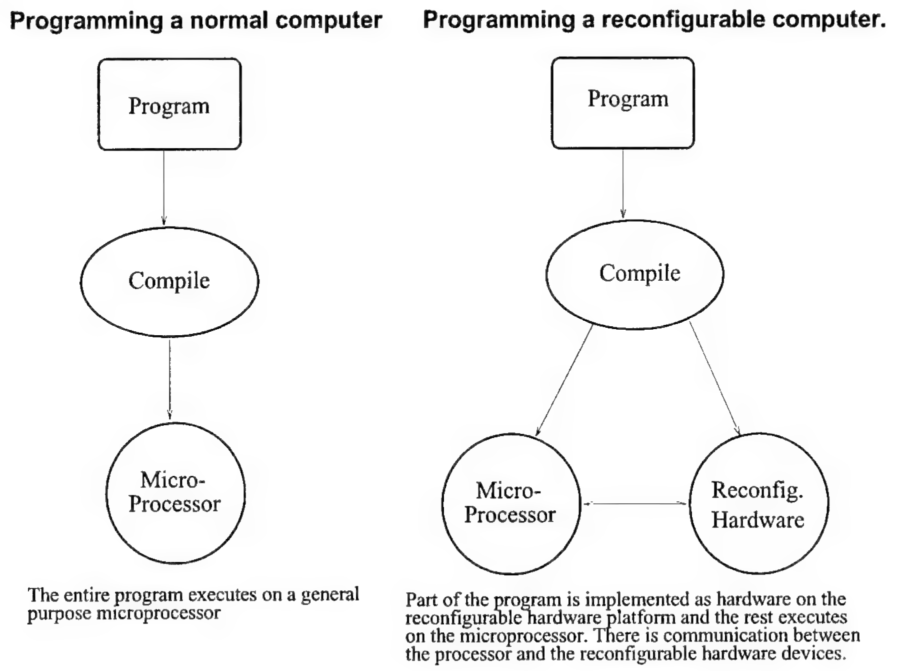

Programming a normal computer
Program
Compile
Micro-
Processor
The entire program executes on a general purpose microprocessor
Programming a reconfigurable computer.
Program
Compile
Micro-
Processor
Reconfig.
Hardware
Part of the program is implemented as hardware on the reconfigurable hardware platform and the rest executes on the microprocessor. There is communication between the processor and the reconfigurable hardware devices.

Figure 3.1: *Programming difference between a general purpose computer and a reconfigurable computer.*

CPU. The software part constitutes the larger portion of the whole program and is compiled for the target workstation. The software portion is usually responsible for writing the configuration `bitstream` into the reconfigurable hardware, providing data to the reconfigurable hardware to operate on and to get back the results from the reconfigurable hardware.

3.2 Taking programs into hardware

The biggest problem in programming a reconfigurable computer is the building of the *hardware* part that executes on the reconfigurable hardware.

Schematic capture methods which have been traditionally used for designing hardware for FPGAs have been found to be very time consuming. In this method the designer thinks of the hardware architecture for his program in terms of register transfers and uses a visual editor (like Viewdraw), to "glue" hardware components (taken from a library) together and implements his hardware design.

With the size and complexity of digital systems growing at an exponential rate, the human mind is quickly reaching the limits of comprehending such systems. So the emphasis is now shifting towards designing at *higher levels* of abstraction where design functionality and tradeoffs are easier to comprehend. Designers can work with fewer design objects at higher levels of abstractions and can think of implementing their designs in a variety of different methods.

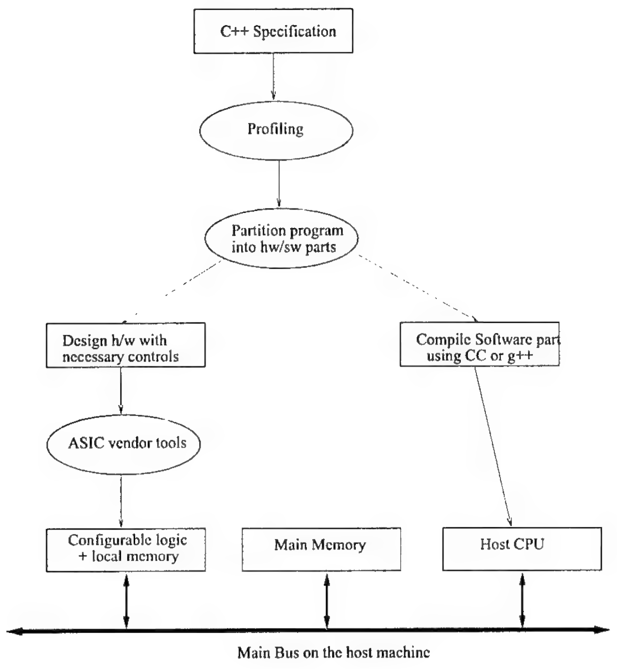

C++ Specification
Profiling
Partition program into hw/sw parts
Design h/w with necessary controls
Compile Software part using CC or g++
ASIC vendor tools
Configurable logic + local memory
Main Memory
Host CPU
Main Bus on the host machine

Figure 3.2: *Design flow on a reconfigurable computer.*

3.2.1 High Level Synthesis

Recently, High-Level Synthesis (HLS) systems are becoming more effective in supporting the design of complex integrated circuits. High-level synthesis converts a technology independent behavioral specification of a digital system into a hardware implementation of the system being described. This design automation at a higher level assures a short design cycle. Described at a high-level, the design is more portable, incremental changes are incorporated more easily, and the design's life cycle is likely to be much shorter.

The high-level synthesis tools take the description of the design as input and generate an implementation of the description using a generic, technology independent models of register-transfer (RT) components. The resulting implementation is a datapath consisting of interconnected instances of generic RT components and a finite state machine describing the controller. Register-transfer level synthesis will then refine this implementation. The generic RT components are mapped to RT components from an existing library, or logic synthesis is used to create a logic implementation of each generic component instance. Figure 3.3 shows the schematic of a high-level synthesis flow.

If the reconfigurable computer systems are to become widely used, good tools need to be developed to help conventional programmers to use them. A reconfigurable logic on a computer would not be of much use to the general software programmers unless there is a way to generate *hardware equivalents* of their software very easily.

The programmers have an option of coding portions of algorithm in an hardware description language like VHDL, but programming in an HDL is quite different from programming in a high-level programming language like C or C++. HDLs have additional constructs to accommodate the intrinsic features of hardware, such as the notions of clocking and asynchrony. Programming in HDL requires the user to think in terms of "hardware" than "software". Even the syntax of most HDLs is quite different from the regular high-level

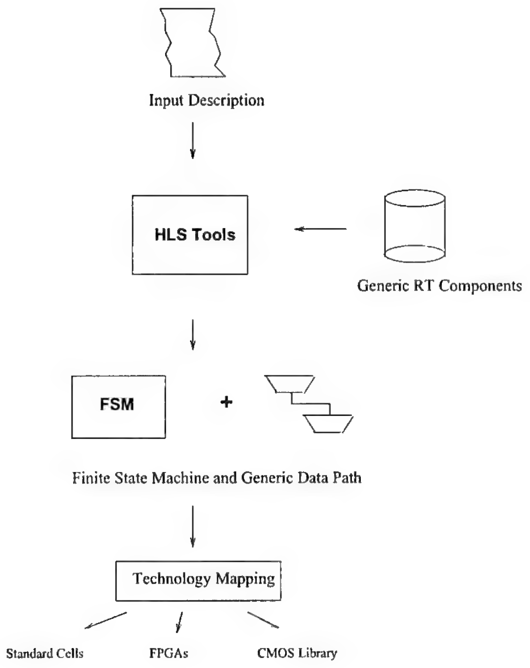


Figure 3.3: *A typical high-level synthesis flow.*

programming languages like C or C++. It would be very convenient to have some sort of a compiler which converts their C/C++ programs "as is" (or with very slight modifications) into hardware.

Since there are already a number of very good commercial tools which compile a VHDL description into hardware very efficiently, we were motivated towards developing a translator for converting C/C++ language programs to synthesizable VHDL descriptions. The translator converts a C/C++ code into a functionally equivalent, *synthesizable* VHDL code. Along with the VHDL translations, it also generates script files for the Synopsys FPGA compiler which helps the user to convert the VHDL code to hardware with minimal interaction.

In order to accomodate the synthesized designs on multi-device reconfigurable architectures like RACE and NEBULA, a CAD tool for partitioning would also be required. The partitioner will take a design and split it into segments such that each segment is of size that doest not exceed the capacity of device(s) on the multi-FPGA boards. The partitioner that we have explained later accomplish these tasks.

For the physical synthesis of designs on FPGAs, CAD tools for placement, floorplanning, and routing are needed. Although there are commercial tools that accomplish this task, their performance and execution times for large designs are not acceptable. Our hierarchical floorplanner addresses these problems by mapping very large designs on large FPGAs in miniscule time without affecting the performance of mapping. In the next chapter, we have described the CAD tools that were developed under the contract.

Chapter 4

CAD Tools

4.1 The DAL C language

It is very difficult to have a translator that translates the whole range of C programs into hardware (or into synthesizable VHDL). C has certain language features which do not have any counterparts in VHDL. Since we use *Synopsys FPGA compiler* to synthesize our VHDL to Xilinx netlist, we also have to consider the limitations of the FPGA compiler and accordingly allow only certain C constructs.

Though VHDL supports `floats`, the *Synopsys FPGA compiler* does not synthesize `floats`.Though floats could have been implemented as lower level functions involving integers, those type of hardware occupy a large number of gates and hence not suitable to be implemented on FPGAs. We should note here that FPGA hardware resources are limited and we need to make the optimum use of this hardware resource. The only C type which is allowed in the language is the `int`. C constructs like `pointers`, `structs` etc. do not make any sense in hardware hence C programs using those constructs cannot be translated into hardware. There is also a restriction not to use any reserved words in VHDL, as variable names, because that would produce a syntactically incorrect VHDL translation. All these and other limitations forced us to define a new language called **DAL C**, which is a subset of the normal C/C++ programming language with some extensions. The extensions allow the user to *read* and *write* into the local RAM on the RACE-I hardware.

In this section we discuss the limitations of the **DAL C** language and describe in detail the syntax of the **DAL C** language.

4.1.1 Restrictions of the DAL C language

We will just mention a few restrictions in this section. Details about the restrictions are also given during the description of the language syntax.

DAL C does not support `floats` or `doubles` for reasons mentioned earlier. It also does not support `char` variables. But any `char` variable can always be represented as an 8 bit integer. Though it does not support `long ints` explicitly, the provision of mentioning the bit length of the variable is given so hardware with more than 32 bit variables can be generated. Character constants (like `'\n'` etc.) are also not supported.

VHDL reserved words cannot be used in writing **DAL C** code. Though the translator translates the program without error, such translations will encounter difficulties when the FPGA compiler is used to synthesize designs. We could have probably mapped them into non-reserved words in VHDL but we have tried to keep this implementation of the translator very simple and not done these mappings. There are also restrictions imposed on the function within a function type of procedure declarations which will become clear in the sections to follow.

DAL C only supports a single-in, single-out type of control flow. This implies that there are no `gotos`, or `return` from functions. There are no equivalent statements in VHDL for the C `gotos`. Since we translate each C function into a VHDL entity with its own `architecture` and do not implement them as VHDL functions the `return` statements are also not allowed. We translate each function into a VHDL entity to generate macro based designs.

C I/O operators like printf(), scanf() etc., are also not supported by the language. These are constructs which access the screen and keyboard and usually the reconfigurable PPGA board usually does not have access to these hardware on the machine.

4.1.2 Types, Operators and Expressions

Variables and constants are the basic data objects manipulated in a program. Declarations list the variables to be used, and state what type they have and perhaps what their initial values are. Operators specify what is to be done to them. Expressions combine variables and constants to produce new values. The type of an object determines the set of values it can have and what operations can be performed on it.

Variable names

There are some restrictions imposed on the names of variables. Names are made up of letters and digits; the first character must be a letter. The underscore "_" counts as a letter; it is sometimes useful for improving the readability of long variable names. Upper case and lower case letters are distinct, so x and X are two different names. Keywords like `if`, `else`, `int`, etc., are reserved: you can't use them as variable names. They must be in lower case.

Also all the reserved words of the VHDL language like `in`, `out`, `port`, etc., should not be used as variable names in the DAL C. Though the compiler will compile the code and translate it to VHDL without any error, the VHDL code thus produced will be syntactically wrong and the FPGA compiler will not synthesize the resulting VHDL code.

Data types and sizes

The only basic data type allowed by the language is the `int`, and the only compound data type is `one dimensional array of int`. A default `int` contains 8 bits, but its size can be changed by a compiler switch and `ints` of custom length can also be created by specifying the size explicitly when declaring the variables or function parameters.

Presently no other data type like the `float` is supported. It is because, we use Synopsys tools to convert our VHDL code into xnf files and as of yet, the Synopsys FPGA compiler does not recognize `floats`.

Constants

Only integer constants are allowed in the language. A `long` as in normal C definition (123456789L), is not allowed and usage of such constants will make the compiler flag an error. Also note that character constants like `'x'`, `'\0'`, etc., which are allowed in normal C are not allowed in DAL C. Use of `enumeration constants` also is prohibited in DAL C.

Declarations

The syntax of the variable declaration in DAL C is very similar to the standard C++ syntax. A variable declaration starts with the optional keyword `static` followed by the keyword `int`. The variable declaration itself consists of its name and, optionally, its width and initial value. If the width is not specified, the compiler assigns the default width 8 or a width specified as an option when the compiler was called. Variables names should confirm to the restrictions imposed in the section on variable names.

All variables must be *declared* before use. A declaration specifies a type, and contains a list of one or more variables of that type, as in

```
int a, b, c;
int a_16bit(16); // not a valid syntax in normal C++
int arr[5];
int arr_9bit(9)[5]; // not a valid syntax in normal C++
```

Variables can be distributed among declarations in any fashion; the lists above could equally well be written as

```
int a;
int b;
int c;
int a_16bit(16);
int arr[5];
int arr_9bit(9)[5];
```

A variable may also be initialized in its declaration. If the name is followed by an equals sign and an expression, the expression serves an an initializer. For example,

```
int a = 10;
int b = 20;
int c = 3;
int a_16bit(16) = 1232;
int arr[5] = { 1, 2, 3, 4, 5};
int arr_9bit(9)[5] = { 1, 2, 3, 4, 5};
```

All variables for which there is no explicit initializer have **undefined random** values.

Arithmetic Operators

The binary arithmetic operators supported are +, -, *. There is an unary -, but no unary +. The + and - operators have the same precedence, which is lower than the precedence of *, which is in turn lower than unary minus. Arithmetic operators, as in C, are grouped left to right. The / operator has to be implemented as repeated subtraction. We do not provide support for the / operator as Synopsys FPGA compiler does not support the / operator in its entirety.

Relational and Logical Operators

The *relational operators* are

```
>    >=    <    <=
```

They all have the same precedence. Just below them in precedence are the *equality operators* :

```
==    !=
```

Relational operators have lower precedence than arithmetic operators, so an expression like `i < lim-1` is taken as `i < (lim-1)`, as would be expected.

The *logical operators* are `&&` and `||`. Expressions connected by `&&` or `||` are evaluated left to right. The precedence of `&&` is higher than that of `||`, and both are lower than the relational and equality operators.

By definition, the numeric value of a relational or logical expression is 1 if the relation is true, and 0 if the relation is false.

Increment and Decrement Operators

The language provides auto increment (++) and auto decrement (–) operators. For example a++ and ++a are equivalent to `a = a + 1`, where as a– and –a are equivalent to `a=a-1`. However an auto incremented or decremented expression cannot be referenced. That is `b = a++;` is illegal. But unlike in C, both ++a and a++ have the same meaning — that of a++.

Bitwise Operators

DAL C provides four operators for bit manipulation.

```
&    bitwise AND
|    bitwise inclusive OR
^    bitwise exclusive OR
~    one's complement (unary)
```

The bitwise AND operator `&` is often used to mask off some set of bits; for example, `n = n & 1`, sets to zero all but the first lowest bit of n. The bitwise OR operator `|` is used to turn bits on. The bitwise exclusive OR operator ^ sets a one in each bit position where its operand have different bits, and zero where they are the same.

One must distinguish the bitwise operators `&` and `|` from the logical operators `&&` and `||`, which imply left to right evaluation of a truth value. For example, if x is 1 and y is 2, then x `&` y is zero while x `&&` y is one.

The unary operator ~ yields the one's complement of an integer; that is, it converts each 1-bit into a 0-bit and vice versa.

Assignments and Assignment Operators

An assignment to a variable modifies the value of the variable, and affects subsequent references to the variable. The formal syntax of an assignment is

```
variable = expression;
```

where expression is either an arithmetic or logical expression.
Expressions such as

```
i = i + 2;
```

in which the variable on the left hand side is repeated immediately on the right, can be written in the compressed form.

```
i += 2
```

The operator += is called an *assignment operator.*

Most binary operators (operators like + that have a left and right operand) have a corresponding assignment operator op=, where op is one of

```
+   -   *   &   ^   |
```

If exprl and expr2 are expressions then,

```
expr1 op= expr2
```

is equivalent to

```
expr1 = (expr1) op (expr2)
```

Quite apart from conciseness, assignment operators have the advantage that they correspond better to the way people think.

Precedence and Order of evaluation

Table 4.1 summarized the rules for precedence and associativity of all operators described above. Operators on the same line have the same precedence; rows are in order of decreasing precedence, so, for example, * has a higher precedence than that of binary + and -. The precedence of both + and - is the same.

Operators	**Associativity**
`()  []`	left to right
`!  ~  ++  --  +  -`	right to left
`*`	left to right
`+  -`	left to right
`<  <=  >  >=`	left to right
`==  !=`	left to right
`&`	left to right
`^`	left to right
`|  &`	left to right
`&&`	left to right
`||`	left to right
`=  +=  -+  *=  &=  ^=  |=`	left to right

Table 4.1: *Precedence and order of evaluation*

4.1.3 Control Flow

The control flow statements of the language specify the order the computations are carried out. We support a single-in, single-out control flow. This implies that no `gotos`, `breaks` from loops, and `returns` are allowed in the language.

Statements and Blocks

An expression such as `x = 0` or i++ becomes a *statement* when it is followed by a semicolon, for example,

```
x = 0;
i++;
k += 10;
```

In C the semicolon is a statement terminator. Braces { and } are used to group declarations and statements together into a *compound statement*, or *block*, so that they are syntactically equivalent to a single statement. The braces that surround the statements of a function are an obvious example; braces around multiple statements after an `if`, `else`, `while` or `for` are another. There is no semicolon after the right brace that ends the block.

If-Else

The `if-else` statement is used to express decisions. Formally, the syntax is

```
if (expression)
  statement1
else
  statement2
```

where the `else` part is optional. The expression is evaluated; if it is true (that is, if expression has a non-zero value), statement1 is executed. If false (expression is zero) and if there is an else part, statement2 is executed instead.

Because the `else` part of an `if-else` is optional, there is an ambiguity when an else is omitted from a nested if sequence. This is resolved by associating the else with the closest previous `else`-less `if`. For example,

```
if (n > 0)
  if (a > b)
    x = a;
  else
    x = b;
```

the `else` goes with the inner `if`, as we have shown by indentation. If that is not what is wanted, braces must be used to force the proper association.

```
if (n > 0) {
  if (a > b)
    x = a;
  }
  else
    x = b;
```

Else-If

A sequence of `if` statements is the most general way of writing a multi-way decision. The syntax of a multiway if statement is

```
if (expression1)
  statement1
else if (expression2)
  statement2
else if (expression3)
  statement3
else
  statement4
```

The expressions are evaluated in order; if any expression is true, the statement associated with it is executed, and this terminates the whole chain. As always, the code for each statement is either a single statement, or a group in braces. The else part handles the "none of the above" or default case where none of the other conditions is satisfied.

Switch

The `switch` statement is a multi-way decision that tests whether an expression matches one of a number of *constant* integer values, and branches accordingly.

```
switch (expression) {
  case const: statements
  case const: statements
  default: statements
}
```

Each `case` is labeled by integer valued constants. If a `case` matches the expression value, execution starts at that `case`. The case labeled `default` is executed if none of the other cases are satisfied. A `default` **must** be present in DAL C. Though cases can occur in any order, `default` should occur at the last. In DAL C, each statement **should** have a `break` statement at the end to indicate an exit from the `switch`. The parser signals an error if the `break` is not included at the end of each statement. For example

```
switch (num) {
    case 0 :
             c = a + b;
             break;
    case 1 :
             c = a - b;
             break;
    case 2 :
             c = a * b; // flags error because no 'break'
    default :
             c = a + b;
 }
```

Loops – While and For

Both `while` and `for` statements support looping. In

```
while (expression)
  statement
```

the expression is evaluated. If it is non-zero, statement is executed and expression is re-evaluated. This cycle continues until expression becomes zero, at which point execution resumes after statement.
The `for` statement

```
for (expr1; expr2; expr3)
  statement
```

is equivalent to

```
expr1;
while (expr2) {
  statement
  expr3;
}
```

Break and Continue

It is sometimes convinient to be able to exit from a loop other thatn by testing at the top or bottom. The `break` statement provides an early exit from a `for` and `while` loop. A `break` causes the innermost enclosing loop or swithch to be exited immediately. `continue` statement is related to `break`, but less often used; it causes the next iteration of the enclosing `for` or `while` loop to begin. In the `while` , this means that the test part is executed immediately; in the `for`, control passes to the increment step.

4.1.4 Functions and Program Structure

The description of a function starts with the optional keyword `void`, followed by the *interface* and the *implementation* of the function or procedure. The interface is composed of the name of the function and the set of input and output parameters, while the implementation is the body of the function. There are two kinds of parameters: parameters passed by value and parameters passed by reference. Parameters passed by value are input parameters, while parameters passed by reference are output only parameters, i.e, the output parameters do not have exactly the same semantics as C++ reference parameters. We had to do this so that the C++ code by itself can be compiled by any C++ compiler, while at the same time, we are able to decide between the input and outputs of the function.

An example of procedure declaration is given below. The `inpA` and `inpB` parameters are 8 bit input parameters and the `outC` is also an 8-bit output parameter. Procedures cannot return a value and thus are of type **void**. This procedure definition gets translated into a hardware macro. Each procedure basically becomes a hardware module. An example of a procedure is given below.

```
void
func_do_this (int inpA, int inpB, int &outC)
{
  // procedure body
}
```

4.1.5 Input and Output - interacting with RACE memory

There is an extension provided by the language that allows the user to interact with the external RACE memory. The language construct uses two functions not there in the normal C/C++ language. They are

```
fread(var, addr);
and
fwrite(var, addr);
```

The *fread()* function gets the value stored in the RACE memory location pointed to by *addr* and stores it in the variable *var*, similarly *fwrite()* writes the value in variable *var* to the address location pointed to by *addr*.

The restrictions imposed here are that *addr* should be an integer variable, it cannot be a constant inside the function. Also the *var* in *fwrite()* cannot be a constant. It has to be an integer variable. If constants need to used it can always be done by assigning a constant to the variable just before the function is called; example :

```
addr = 10;
fread(var, addr);
or
var = 40;
addr = 20;
fwrite(var, addr);
```

4.2 Multi-Way Partitioning for Programmable Board Architectures

This section describes the multi-FPGA based partitioning. Section 4.2.1 gives a brief introduction to our method of board level partitioning. Section 4.2.2 gives a formal definition to the multi-way partitioning problem. Section 4.2.3 gives a brief overview of the partitioning approach used in our work. In section 4.2.4 we describe the pre-processing step used for synchronous sequential designs. Sections 4.2.5, 4.2.6, 4.2.8, 4.2.7 illustrate the algorithms used for each step involved in our partitioning approach.

4.2.1 Introduction

Our approach is timing driven in order to improve the clock speed of the partitioned design. The partitioner addresses both flat (gate-level) and macro based designs. Macro based designs not only speed up the partitioning process but also give better performance in terms of timing. Since the number of modules on the top level are lesser the partitioning problem becomes easier and it allows the user to handle extremely large circuits.

We use *path based clustering* to capture the connectivity of the modules and timing on critical circuit paths in order to reduce the *cutset* during partitioning and improve the clock speed of the partitioned design. This is followed by *k-way* partitioning for forming multiple partitions. This is followed by *pin-assignment* and *global signal routing* for all the *cut signals.* The delays introduced on the circuit paths because of being cut during partitioning are incorporated into the circuit delay model to give a better estimation of clock speed during retiming. Retiming is used to improve the clock speed of the partitioned design.

4.2.2 Problem Formulation

The multi-way partitioning problem under architectural constraints can be formulated as follows :
Given : A circuit $\mathcal{C}$ represented as a hyper-graph, an MFS architecture $\mathcal{A}$ with some $\mathcal{D}$ number of devices. $\mathcal{C} = (V, E)$ where $V = \{v_1, v_2, \ldots v_n\}$ and $E = \{e_1, e_2, \ldots e_m\}$ where $e_i \subset V, 1 \leq i \leq m$. Let $\mathcal{S}_j$ = size of device j, $1 \leq j \leq \mathcal{D}$, and let $\mathcal{P}_j$ = # of pins on device j, $1 \leq j \leq \mathcal{D}$. For our application, $\mathcal{S}_i = \mathcal{S}_j\ \forall i, j$ and $i \neq j$, and $\mathcal{P}_i = \mathcal{P}_j\ \forall i, j$ and $i \neq j$.
Objective: Partition $\mathcal{C}$ into less than or equal to $\mathcal{D}$ segments such that the following conditions are satisfied,

- Size of each segment j is less than or equal to $\mathcal{S}_j$, $1 \leq j \leq \mathcal{D}$.
- *Cutset* for each segment j is less than or equal to $\mathcal{P}_i$, where *Cutset* is the set of nets E_j such that at least one node for each net in E_j is not is segment j. Ideally, it is desirable to have $Cutset(j, k) \leq interconnect(j, k)$ where we define $Cutset(j, k)$ as the number of nets that have at least one node in segment j and at least one node in segment k, and $interconnect(j, k)$ as the width of the board level interconnection between the device embedding the segment j and the device embedding the segment k. If the later condition is not satisfied, the partitioned solution would not be able to route all the cut signals using the direct interconnect between segments $j\&k$, $1 \leq j, k \leq D, j \neq k$. Our partitioner tries to find a direct routing $j \rightarrow l, l \rightarrow k, j \neq k \neq l, 1 \leq j, k, l \leq D$.
- User defined timing constraints are satisfied[1].

4.2.3 The Board-Level Partitioning Approach

The partitioner deals with the partitioning of VLSI circuits specified in terms of gates or macros[2]. The partitioning is performed under board-level and user-defined parameters and constraints, see figure 4.1. The board level constraints are defined in the $RACE - I$ architecture description. The architectural description defines the pin-connectivity among the FPGAs, the board level delays and CLB delays of the chosen Xilinx FPGA architecture and the connections to the controller and the FPGA's local memory. The board-level partitioning involves the mapping of the input design onto the four FPGAs on the board.

The multi-way cut method used is shown in figure 4.2. The *k-way* partitioning proceeds in the *clock-wise* direction starting with FPGA# 1 and terminating with FPGA# 4, see figure 4.2.

Our partitioner addresses only *module* partitioning. The *memory* partitioning problem is not addressed by our partitioner. The memory usage of the design can be determined only after design simulation. It cannot be determined by the partitioner.

4.2.4 Preprocessing for Sequential Designs

A synchronous sequential circuit consists of clocked flip-flops and combinational logic elements connected together by a feedback path. An algorithm suggested by Murgai breaks the feedback loops at the input of all

[1]The partitioner accepts user defined constraints as an input in form of the application support file. The timing constraints can be declared in the application support file as *global signals* and *critical paths* in the design. For example, signals like CLK or GLOBAL RESET may have dedicated lines on the board to route them. In such a case these signals can be prevented from being cut and being routed over the programmable interconnect for performance optimization by declaring them as *global signals* in the support file. The *critical paths* are prevented from being cut during partitioning. This again helps in improving the timing performance of the design after partitioning. In addition to these constraints, the partitioner uses a *timing driven* clustering approach (see section 4.2.5) to minimize the worst case delay in the partitioned circuit. The partitioner also uses *Retiming* in order to improve the clock speed of the partitioned design.

[2]Macros are circuit blocks which are themselves a group of connected gates implementing a certain function.

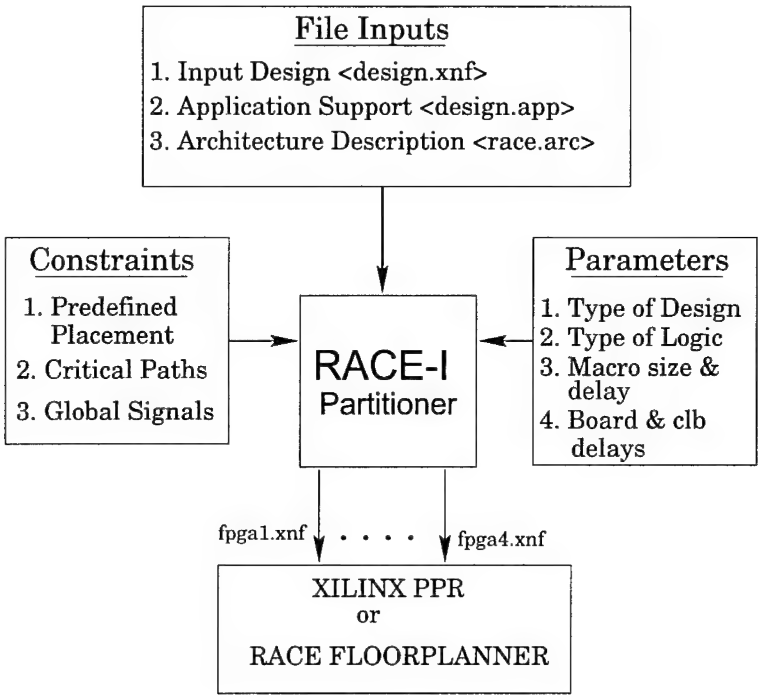


Figure 4.1: *Partitioner Overview*

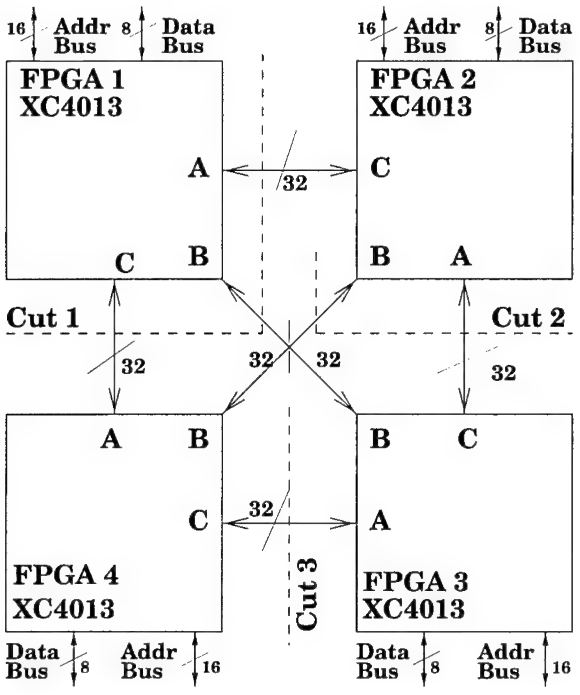


Figure 4.2: *The Multi-Way Cut method*

Algorithm 4.2.1 Conversion to DAG

```
Input: Directed Cyclic Graph ($DCG$) of the design netlist file.
Output: Directed Acyclic Graph ($DAG$) of the design netlist file.
    { Call Depth_First_Search($graphG$);
    For each vertex in the graph G
        List vertices in decreasing finishing times in L;
    Perform Transpose($G$);
    While (List of vertices L is not empty)
        Call Depth_First_Search($G_T$);
    Output vertices in each tree in depth-first forest as strongly connected components;
    }
```

flip-flops. A depth first search algorithm is run on the DCG and the start and finish time stamps are noted for each node. A list L of the all the vertices of the graph G in the order or decreasing finishing times is formed. A *transpose* of the given graph is obtained by reversing the direction of signal flow in the edges of the graph. A depth first search is run on this *transposed graph* considering the vertices in the order defined in list L. The vertices of each tree in the depth first forest are given out as *strongly connected components.* The nodes in a cycle are considered as a single vertex to form an acyclic graph representation. This preprocessing step is done for synchronous sequential designs in order to convert the given circuit graph to a *DAG*. The method is presented in Algorithm 4.2.1.

4.2.5 Clustering Method

Clustering boosts performance and improves the quality of partitioning. Our objective function for clustering is both connectivity and timing driven.

Path-based clustering is done to reduce the number of cuts on the circuit paths. A depth-first(DF) traversal is done on the circuit graph starting with the input nodes of the given graph. A `source set` is formed which contains the input nodes of the graph. For each of these sources we do a depth first traversal. As we proceeds, we select a node to be placed in the cluster depending on its area, connectivity, and delay. Once the traversal for all the sources is completed, all the nodes have been assigned to clusters. Our heuristic aims to reduce the number of *path cuts* in the circuit by traveling along the circuit paths while clustering the nodes. The size of the clusters formed during clustering, in terms of their area and associated pin sizes affects the satisfiability of the area, pin, and timing constraints during partitioning.

S is the set of all previously clustered nodes of the current cluster, *clusterArea* is the area of the present cluster. $clusterSize$ is the maximum specified size of the cluster taken as an input from the user. $area(u)$ is the area of a node $u \in G$. E_u is set of nets incident on a node u where u is the node being visited during traversal, $w(e_j)$ is the weight of any net $e_j \in E_u$, $w(e_j) = \sum_{k \in e_j} area(k)$, where $area(k)$ is the area of node k, $w(u)$ is weight (size) of the node u, $w(v)$ is weight of a clustered node v to which the net e_j is connected, The weights α and β are *empirically* chosen to be 1/2 and 1 respectively. Using α = 1/2 gave good approximation for the attraction of the given node to unclustered nodes. Values of α ranging from 1/4 to 2 were tried out and a value of 1/2 was seen to give the best results (in terms of *reduction in cutset*) for most of the benchmark circuits used in our work. The *delay* metric varies linearly with the *worst case* delay of the node and so choosing a value of 1 for β reflects this variation appropriately. Values of β ranging from 1/4 to 2 were tried out and a value of 1 was found to give the best results (in terms of *reduction in worst case*

Algorithm 4.2.2 Path Based Clustering

```
**Input:** Directed Acyclic Graph G
**Output:** A group of clusters on the input graph G
Mark all the nodes in the graph as UnVisited;
Calculate the path delay values of each node in the circuit graph using a topological ordering
of the nodes;
Form the Source Set $Sources = \{source_1, source_2, \dots, source_l\}$ where $source_i$ is a primary
input node of the design and l is the number of input nodes;
while ( $Sources \neq \Phi$ & all nodes are not marked Visited )
    { Make a depth first traversal of the circuit graph for $source_i$;
    Calculate the cost of adding a node $u \in G$, where u belongs to the adjacency list of
    nodes in the present cluster C, to the cluster C using the attraction function;
    Add the node with the maximum attraction and satisfying area constraint
    $(area(u) + clusterArea \leq clusterSize)$ to the present cluster;
    If area constraint is not satisfied start a new cluster starting with the node u;
    Mark the node as Visited and update its cluster # and the clusterArea;
    If all nodes have been traversed for the $source_i$
    $Sources = Sources \setminus source_i$;
    $i \leftarrow i + 1$; }
```

path delay) for most of the benchmark circuits. The heuristic is presented as a *pseudocode* in Algorithm 4.2.2.

4.2.6 Multi-way Partitioning

Fiduccia-Mattheyses algorithm (FM) based min-cut k-way partitioning is performed on the given design to produce multiple partitions. FM is applied repeatedly on the given circuit. After each partition, a slice is removed from the original circuit. The remaining circuit is once again partitioned using FM (this time on the reduced circuit size). This is continued until the remaining circuit is small enough to fit in a single FPGA. Once this stage is reached we have multiple partitions each of which has to be embedded into a FPGA. *CLIP* is used to reduce the number of signals cut by moving natural clusters to one partition. We will look at how the basic FM bi-partitioning heuristic works. Then we will look at how CLIP helps in reducing the cutset. The FM heuristic is used to find a solution to the following bi-partitioning problem:

Given : a circuit *C* consisting of *n* cells connected by a set of *m* nets, the problem is to partition circuit *C* into two blocks *A* and *B* such that the number of nets which have cells in both the blocks is minimized and the balance factor $r(= \frac{|A|}{|A|+|B|})$, where $\mid A \mid$ and $\mid B \mid$ are sizes of partitioned blocks *A* and *B* respectively, is satisfied.

A *pseudocode* for the modified FM heuristic is presented in Algorithm 4.2.3. In Algorithm 4.2.3,

- '*designSize*' : is the size of the current input design in terms of CLBs. As the k-way partitioning proceeds, the size of the design goes on decreasing.
- '*fpgaSize*' : is the size of the given FPGA size as defined in the architecture description file in terms of CLBs.

- '*Gain of a cell*': The gain $g(i)$ of a cell 'i' is the number of nets by which the cutset would decrease if cell 'i' were to be moved. A cell is *moved* from its current block to its complementary block.
- '*Balance criterion*': To avoid having all cells migrate to one block a balancing criterion is maintained. A partition (A, B) is balanced if

$$r. \mid V \mid -s_{max} \leq \mid A \mid \leq r. \mid V \mid +s_{max} \tag{4.1}$$

  where $\mid A \mid + \mid B \mid = \mid V \mid$; and $s_{max} = Max[s(i)], i \in A \cup B = V$.
- '*basecell*': The cell selected for movement from one block to another is called '*basecell*'. It is the cell with the maximum gain and the one whose movement will not violate the balance criterion.
- '*pass*' : is the counter for the number of iterations.
- '*freecellset*' : is the set of cells which have not been locked in the current pass of FM.
- '*endofFM*' : is the flag which indicates the end of FM, when FM can no longer reduce the cutset.
- '*endofPass*' : is the flag which indicates the end of the present pass, when the cutset can no longer be reduced during the pass.
- '$cutsize_{pass}$' : is the cutset size at the end of the current pass.
- '$cutsize_{pass-1}$' : is the cutset size at the end of the previous pass.
- '$C = \{c_1, c_2, \ldots, c_n\}$' : is the set of all cells which have not been assigned partitions and n is number of cells which have not been Assigned partitions.
- '$m_1, m_2, \ldots, m_k$' : are the move numbers.
- '$area_1, area_2$' : are the areas of the two partitions.

In the FM method, cell gains are calculated based on the immediate benefits of moving cells. After a cell is moved, the gains of its neighbors are updated. At any stage in the move process, the *total* gain of a cell can be broken down as the sum of the initial gain component and the updated gain component. The total gain indicates the overall situation of the cell, while the updated gain component reflects the change in the cell's status due to the movements of its neighbors. Instead if cell movement decisions are based primarily on their updated gain components, the distraction during the cluster pulling effort caused by cells not in the cluster currently being moved is reduced. It allows the heuristic to concentrate on a single cluster at a time for moves in one direction. The initial gain of the cell is, however, useful for choosing a starting seed. For implementation sake, the cell gains are set to zero after the initial gain calculation, but they are ordered according to their initial gains. This is shown in algorithm 4.2.3 as, "*make gains of all cells = 0 after 1st cell selection*". Cell gains are updated as in the original FM method. The advantage of this method is that the clusters lying entirely in one subset can be easily moved to the other subset. This is because cells gains being cleared to zero in the initial stage causes cells in a cluster to have less inertia in staying inside their original subset. The benefit of this movement is that larger but less densely connected clusters can be removed from the cutset by moving their densely connected constituent clusters from one subset to the other.

Algorithm 4.2.3 (K-way Partitioning)

```
while (designSize $\leq$ fpgaSize) {
    while (endofFM $\neq$ TRUE) {
        $pass \leftarrow 1$;
        $C = \{c_1, c_2, \ldots, c_n\}$;
        for each $i$, $1 \leq i \leq n$, compute gain ($g_i$);
        freecellset $F \leftarrow C$;
        while (endofPass $\neq$ TRUE) {
            Select cell with highest gain which does-not violate constraints;
            make gains of all cells = 0 after 1st cell selection;
            If no basecell found endofPass = TRUE;
            lock basecell;
            update partition areas($area_1, area_2$);
            $F \leftarrow F - basecell$;
            update gains of all cells connected to basecell;
            }
        Select best sequence of moves $m_1, m_2, \ldots m_k (1 \leq k \leq i)$
                such that $G = \sum_{j=1}^{k} g_j$ is maximum;
        if $G < 0$ exit;
        calculate $cutsize_{pass}$;
        if $cutsize_{pass} = cutsize_{pass-1}$ endofFM = TRUE;
        Make all $k$ moves permanent;
        Free all cells;
        $pass \leftarrow pass + 1$;
        }
    $designSize = designSize - area_1$; }
```

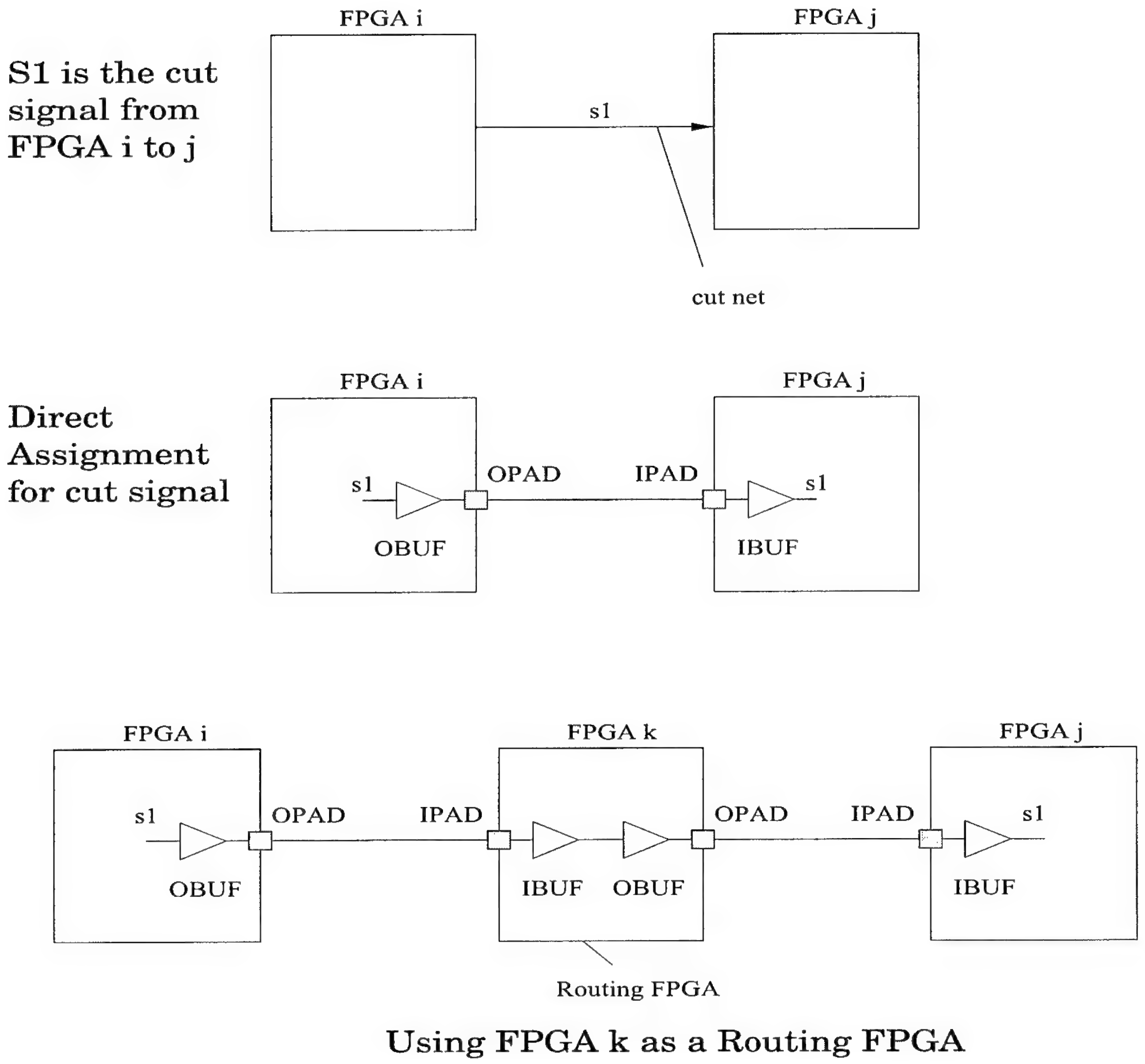


Figure 4.3: *Pin Assignment for a Cut Signal*

4.2.7 Pin Assignment

In a Multiple FPGA System, while the FPGAs themselves can be routed and re-routed, the wires moving signals between FPGA pins are fixed by the routing structure on the implementation board. While impressive results have been achieved by hand-mapping of algorithms and circuits to FPGA systems, developing a completely automatic system for mapping to these structures is important for greater utility of these systems. For global routing purposes, the multi-FPGA system can be thought of as a complex graph including internal resources or as an abstract graph with FPGAs as nodes. Various solutions have been attempted in the past for the pin assignment problem:

- One solution is to ignore the problem. After Global Routing has routed signals through intermediate FPGAs, those signals are then randomly assigned to individual pins.
- A specific topology can be used for simplifying the problem. Topologies such as bipartite graphs are used which connect logic-bearing FPGAs with routing-only FPGAs. The logic-bearing FPGAs are placed initially and it is assumed that the routing-only FPGAs can address any possible pin assignment. This applies to topologies such as bipartite graphs and partial crossbars, where logic-bearing FPGAs are not directly connected. This is not our case.
- The FPGA placement tool determines its own assignment. The user is allowed to restrict the locations where an I/O pin can be assigned. With such a system, I/O signals are restricted to only those pin locations that are wired to proper destinations. Once the placement tool determines the pin assignment for one FPGA, this assignment is propagated to the attached FPGAs.

In our work, pin assignment, refer figure 4.3, is done based on the *Architecture Description* as given in the architecture description file (for the $RACE-I$ architecture, the file is called `race.arc`). The file defines the connectivity between the FPGAs on the board. From the connectivity information an architecture model is formed and this model is used for pin assignment and global signal routing. The model is an abstract model of the MFS, where each FPGA is treated as an entity. The architecture description defines a specific connection available for each pin on the board so the problem is very restricted. The procedure used for I/O pad assignment is presented in Algorithm 4.2.4.

A set of all cut nets,

$$CutNets = \{C_1, C_2, \ldots, C_n\} \tag{4.2}$$

is formed, where C_i is a cut net and n is the total number of cut nets. For each cut net, depending on the FPGAs in which the modules on this signal are, pads are assigned on the FPGAs according to the connectivity information. If the number of cut signals between any two FPGAs is greater than the width of interconnect between the FPGAs then a direct assignment for the cut signal cannot be made. In such a situation available pins on other FPGAs are used for completing the routing. In such a case the other FPGAs act as *routing FPGAs* only. Such a routing introduces ≥ 2 board-level delays and delay due to routing inside the extra FPGA which is a great penalty in terms of delay for that cut signal.

The pin assignment phase is followed by the *retiming*. The necessary buffers and pads are introduced into the design and this information is written into the individual FPGA design netlist files after retiming is performed on the design[3]. This produces a feasible mapping of the partitioned design on the board. These netlist files can be directly given as an input to the $RACE-I$ floorplanner or the Xilinx PPR tools.

[3]This is because the pin assignment phase is done to find out the exact mapping of the cut signals on the $RACE-I$ architecture. Once retiming is done there are changes in the design netlist (please refer to section 4.2.8 for details) which have to be written into the design netlist files for the individual FPGA files. However, the number of cut signals is not changed by retiming

Algorithm 4.2.4 (IOAssignment)
Input: `CutNets` *which contains all the nets that are cut and the modules that are connected to the cut net. It also contains the FPGA in which each module is present.*
Output: *Design file for each partition with iopads and buffers included for cut nets.*

```
{
    for every net in the cut nets set do
        {
            SOURCE = source_cell_FPGA#;
            SINK = sink_cell_FPGA#;
            Try to find a direct routing between the FPGAs;
            if found
              { Assign Pads for the net on both FPGAs;
              assigned_flag for the net = TRUE;
              Write the iopad information to the design file; }
            else
              assigned_flag for the net = FALSE;
        }
    for every net in the cut nets set do
        {
            if assigned_flag is FALSE
              { Try to find a valid routing (possibly with minimum hops)
              through available pins on other FPGAs;
              If valid routing found
                { assigned_flag for the net = TRUE;
                Write the I/O information to design file; }
              else
                { assigned_flag for the net = FALSE;
                write net information to unRouted file; } }
        }
}
```

4.2.8 Retiming

The pin assignment and global signal routing is followed by *Retiming*. We will look at the formal definition of *retiming* and the various steps involved in retiming which will be followed by the procedure used by our partitioner in implementing *retiming*.

Retiming is a circuit transformation in which registers are added at some points in the circuit and removed from others in such a way that the functional behavior of the circuit as a whole is preserved. *Retiming* is used to reduce the combinational rippling in the circuit thereby improving the clock speed of the circuit.

For understanding how the clocked circuits are optimized by relocating registers, we use the correlator circuit shown in figure 4.4. A correlator circuit takes a stream of bits $x_0, x_1, \ldots$ as input and compares it with a fixed-length pattern $a_0, a_1, \ldots, a_k$. After receiving each input x_i, the correlator produces as output the number of matches given by:

$$y_i = \sum_{j=0}^{k} \delta(x_{i-j}, a_j) \tag{4.3}$$

where δ is a comparison function, such that, it is 1 if $x = y$ and 0 otherwise.

A *synchronous circuit* is a circuit which satisfies all the above restrictions. A *re-timing* of a circuit $G = \langle V, E', d, W \rangle$ is an integer-valued vertex-labeling

$$r : V \rightarrow Z \tag{4.4}$$

Retiming changes the original circuit to a new circuit $G_r = \langle V, E', d, w_r \rangle$, where the edge-weighting w_r is defined for an edge $u \overset{e}{\rightarrow} v$ by

$$w_r(e) = w(e) + r(v) - r(u) \tag{4.5}$$

For any retimed circuit the conditions laid down by Theorem 1 should be satisfied. Theorem 1 provides the basic tool needed to solve the clock-period minimization problem. A legal re-timing of the circuit is obtained only when the retimed circuit satisfies the conditions laid down by Theorem 1.

The minimum feasible *clock period* for a synchronous circuit G, is given by the maximum amount of propagation delay through which any signal must ripple between clock ticks. The clock period is defined by

$$\Phi(G) = max\{d(p) : w(p) = 0\}$$

4.3 Floorplanning

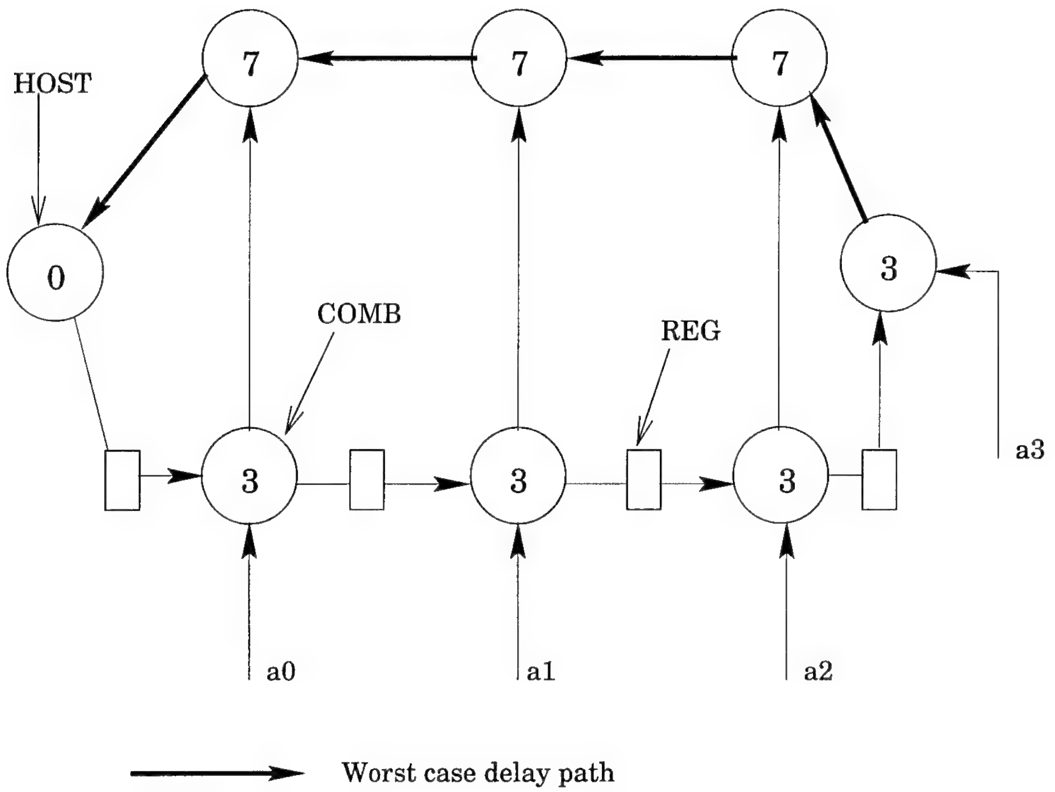
In this section, we describe the work related to floorplanning. First, we describe the problem addressed in this work. Then, we go on and present a detailed overview of our solution of the problem. Section 4.3.1 provides a detailed description of the floorplanning problem for FPGAs. Section 4.3.3 describes solution methodology. Section 4.3.4 to 4.5 describe the detailed steps of our solution.

4.3.1 Problem Formulation

We address the problem of floorplanning a macro based design for FPGAs. Macros are a collection of relatively placed CLBs. They can be *hard* (of fixed relative placement of CLBs, hence fixed shape), or *soft* (of flexible relative placement of CLBs, hence flexible shape). The goal is to decide dimensions of macro blocks, and allocate locations to them on the target FPGA chip such that no part of the chip is allocated

BEFORE RETIMING

Clock Speed before Retiming = 24 esec



COMB	A combinational block
REG	A sequential block (flip flop)
HOST	The host for the circuit

Figure 4.4: *A Correlator Circuit Before Retiming*

Theorem 1 *Let G=* $\langle V, E', d, w\rangle$ *be a synchronous circuit. Let c be an arbitrary positive real number and let r be a function from V to integers. Then r is a legal re-timing of G such that* $\Phi(G_r) \leq c$ *if and only if*
$r(u) - r(v) \leq w(e)$ *for every edge* $u \xrightarrow{e} v$ *of G, and*
$r(u) - r(v) \leq W(u, v) - 1$ *for all vertices* $u, v \in V$ *such that* $D(u, v) < c$.

to more than one macro. In the process of physical mapping of the macros, we try to minimize the area occupied by the floorplan, while also trying to minimize the total interconnection length and the length of longest interconnect in the floorplanned design.

4.3.2 Input-Output

Following are the inputs to the floorplanner that are also illustrated in figure 4.5 :

1. **Design files :** The floorplanner takes macro based designs as input.The design is given as a set of files in xilinx netlist format (xnf). The set of files includes :
    - *Top level xnf file :* It gives the interconnection pattern of macros.
    - *Macro xnf files :* An xnf file for every type of macro used in the design.
2. **Support files**
    - *Macro description file :* It contains information about all macros used in the design.
    - *Architecture description file :* It Contains information about FPGA chips used as target architecture for floorplanning.
3. **Target FPGA :** The FPGA chip, on which the input design should be floorplanned.
4. **Degree of Compactness :** Degree of compactness governs area of the floorplanned layout. Its range is 0 to 2. Degree of compactness equal to 2 results in minimum area.

Outputs of the floorplanner are :

1. **Constraint file :** It gives starting locations of all the macros.
2. **Updated macro xnf files :** Floorplanner reshapes soft macros in the input design. These are the xnf files for these reshaped macros.
3. **Updated top level xnf file :** This file gives interconnection pattern between the macros. The macros included in this file are the reshaped macros.

The constraint file and the set of xnf files output by the floorplanner are presented to Xilinx router for circuit routing. CAD flow involving the floorplanner is illustrated in figure 4.6.

4.3.3 Overview of Methodology

Basic approach, we have adopted to address the problem is *min-cut based floorplanning.* First we divide the design in multiple segments, each containing a group of macros, and place these segments on the target FPGA chip. Then, in each of these segments, we reshape the constituent soft macros and allocate exact locations to all the macros. Following which, we go on to perform intra macro placement for soft macros. Finally we do compaction on the floorplan to minimize the area occupied by the floorplan. Overview of the approach to floorplanning is shown in the Figure 4.7. Floorplanning is accomplished in the following stages :

- *Successive Bipartitioning* : This step divides the input design into multiple segments, and assigns area on FPGA for each segment. Each of these segments is essentially, a set of macro blocks.

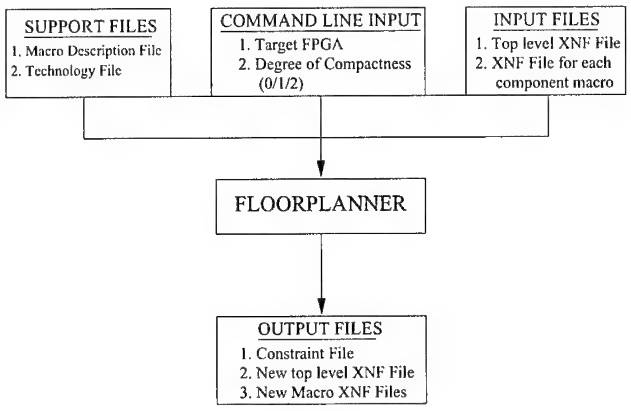


Figure 4.5: *Input Output interface of the floorplanner*

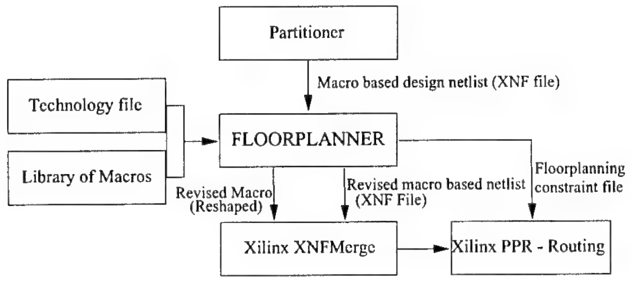


Figure 4.6: *CAD flow for using the floorplanner*

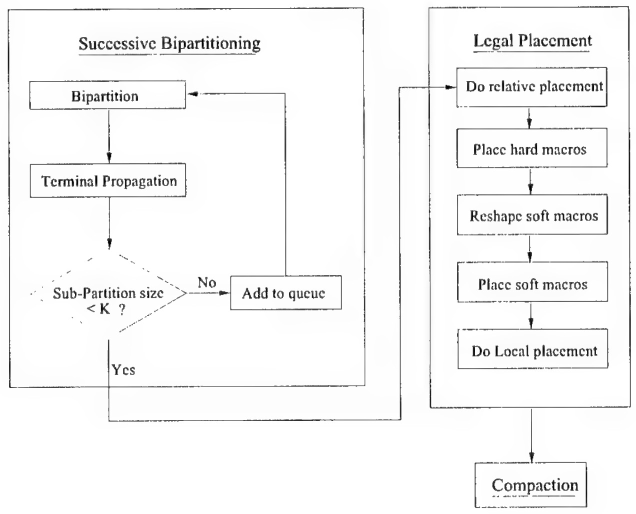


Figure 4.7: *Flow of execution of the floorplanner*

- *Legal Placement* : This step operates individually on every group of macro generated by successive bipartitioning. In each segment, we assign exact position to every macro, and to the CLBs inside each macro. We also reshape soft macros. We try to place macros using minimum area in each group. Basic floorplan layout of the input design is obtained in this step.
- *Compaction* : Compaction operates on floorplanned layout, as given by the previous step. During compaction, we try to minimizc the area occupied by the floorplan by eliminating unused CLBs within the bounding box of the floorplanned layout.

In the rest of the chapter we provide detailed description of various steps of our solution. Section 4.3.4 explains the process of successive bipartitioning in detail. Legal placement step is illustrated in section 4.4. Section 4.5 describes global compaction done on the floorplan. Finally, we conclude with a summary.

4.3.4 Successive Bipartitioning

Following definitions will help in understanding the process :

Bipartition : Given a set, `Part1`, of n modules $M_1, M_2, M_3, ..., M_n$ of sizes $S_1, S_2, S_3, ..., S_n$ respectively, *bipartitioning* is a process of dividing the modules in two sets, `SubPart1` and `SubPart2` of nearly equal sizes.

Partition Segment : Set of modules generated by *bipartitioning* is called *partition segment.* A bipartitioning of a set generates two *partition segments.*

Segment Number : Each *partition segment* is assigned a unique number, called *segment number.*

Parent Partition Segment : A partition segment is *parent* to all the modules contained in it.

Cardinality of Partition Segment : Number of modules in the segment is referred to as its *cardinality.*

Cutline : All partition segments are mapped to a specific area on FPGA chip. Along with each bipartitioning of partition segment, corresponding area on FPGA chip is also divided in two blocks by a *cutline.* *Cutline* can be a *horizontal cutline* or a *vertical cutline.* Ratio of the area of blocks is same as the ratio of sizes of partition segments generated by the bipartition.

Area Slice : Every block of area on FPGA chip generated by *cutlines* is called an *area slice.* Each *partition segment* is mapped to one and only one *area slice*, and each area slice has one and only one partition segment mapped to it.

Successive bipartitioning is a process of dividing the design into multiple segments, such that cardinality of each segment is less than or equal to a constant, K. We start with the input design as initial partition segment and assign it to whole of the FPGA area. We continue to *bipartition* the segments, until the terminating condition (*cardinality* of each *partition segment* equal to K) is satisfied. With every bipartitioning, we also divide the area of FPGA chip, allocated to that segment, in two *area slices.* Then, we allocate each new *partition segment* to an *area slice.*

The process of successive bipartitioning is described in detail in the following. A queue `To_Cut` is maintained to keep track of partition segments, which are candidates for further partitioning. A partition segment is candidate for further bipartitioning if and only if, number of macros in it is greater than a constant K. The queue is initialized by the original design. In one pass of successive bipartitioning, head of the

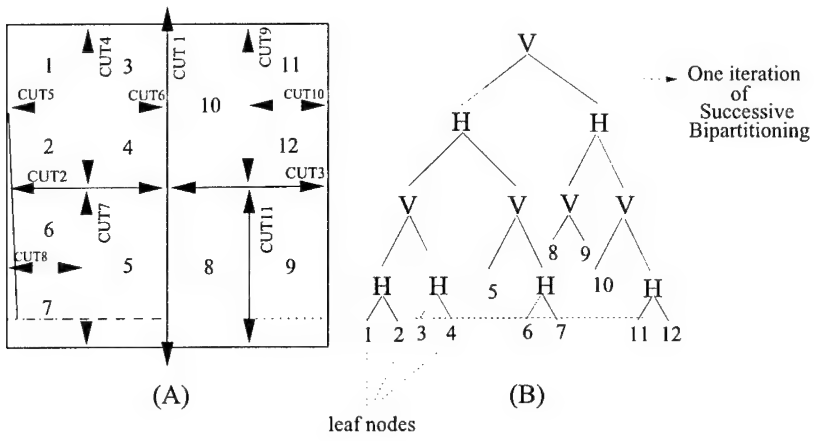


Figure 4.8: *(A)-A possible order of cuts, (B)-Corresponding partition tree*

queue `To_Cut` is bipartitioned. At the same time, area of the target FPGA is sliced in two parts by a *vertical cut,* and each *area slice* is allocated to a segment obtained as the result of bipartitioning. Out of these *partition segments,* eligible candidates for further partitioning are bipartitioned. The corresponding areas on the target FPGA are now sliced by a *horizontal cut.* Mapping of the new subpartitioned segments to sliced area is done. Segments, which are eligible candidates for further bipartitioning, are added to the queue `To_Cut`. This process is repeated until the queue `To_Cut` is not empty. This process of iterative bipartitioning effectively forms a *partition tree* whose nodes are *H or V* indicating horizontal, or vertical cut respectively, and leaf cells are the partition segments with less than or equal to K number of modules. Each of these leaf cells have an *area slice* assigned to it on FPGA. A possible order of cuts is shown in Figure 4.8 (A), and the corresponding partition tree is shown in Figure 4.8 (B). $Grid_{xy}$ is a two dimensional integer array, which maps the CLB slots of the FPGA to CLBs of the design being floorplanned. Integer value at the index (x, y) in this array, corresponds to the segment number occupying that slot on the FPGA. After completion of successive bipartitioning $Grid_{xy}$ gives the mapping of each segment of the design on the FPGA. Since, all the leaf nodes of the partition tree are mapped to the array $Grid_{xy}$, the physical area of the FPGA chip allocated to the tree nodes is determined. Each of these nodes contain less than or equal to K number of macros. Hence, effectively, mapping of groups of macros to localities on the FPGA is obtained.

Bipartitioning is done using Fiduccia Mattheyses (FM) method. Initial cut for FM is generated by clustering the input set of macros in two clusters. Following partitioning, terminal propagation is done to keep global knowledge of connections in each segment.

Partitioning of the circuit is done with cutset [4] as a constraint. This places densely connected modules in one partitioning segment. We try to generate equal sized partition segments by every bipartitioning, but it is not always achievable because of difference in size of macros. The actual ratio of size, in which the bipartitioning occurs is given by *Ratio.* After completion of this step, following holds true

$\forall Segment_i \mid 1 \le i \le NumberofSegments, Cardinality(Segment_i) \le K$

The process of successive bipartitioning results in relative placement of groups of macros. Depending on the number of cuts during hierarchical bipartitioning, the depth of partitioning tree can vary. The leaf cells of partitioning tree should contain less than or equal to a constant number of modules. Successive

[4]Cutset is defined as the number of nets connecting macros across two partitions

Algorithm 4.3.1 (Successive-Bipartition Algorithm)
SuccessiveBipartition()

```
{   To_Cut ← $Source\_Part$;
    while(To_Cut $\neq \phi$)
    {   $CutPartition \leftarrow Head($To_Cut$)$;
        DoPartition(CutPartition, SubPart1, SubPart2);
        $Count \leftarrow$ Cardinality$(SubPart1)$;
        if($Count \geq 4$)
        {   DoPartition(SupPart1, SubPart3, SubPart4);
        }   PutInTo_Cut(SubPart3, SubPart4);
        else $Legal\_Place \leftarrow Legal\_Place + SubPart1$;
        $Count \leftarrow$ Cardinality$(SubPart2)$;
        if($Count \geq 4$)
        {   DoPartition(SupPart2, SubPart3, SubPart4);
        }   PutInTo_Cut(SubPart3, SubPart4);
        else $Legal\_Place \leftarrow Legal\_Place + SubPart2$;
    }
}
```

bipartitioning is followed by *Legal Placement*, where we assign physical location to every macro.
Steps for Successive Bipartitioning are illustrated in the following subsections.

Clustering

The algorithm given below describes the flow of clustering. It addresses the following problem :

Given a circuit C consisting of n cells connected by a set of m nets, the problem is to make two clusters A and B out of the circuit C such that the number of interconnections between the two clusters is minimized and the ratio r , where $r(=\frac{|A|}{|A|+|B|})$, is satisfied. Here $|A|$ and $|B|$ are sizes of clusters A and B respectively.

Let $p(i)$ be the number of pins of cell i and $size(i)$ be the size of cell i, $1 \leq i \leq n$. The following definitions are useful in explaining the procedure.

Size of cell: The size $size(i)$ of cell i is the number of CLBs in the cell.

Size of Cluster: The size $Size_i$ of cluster i is the summation of sizes of all the modules in the cluster. If there are X modules in a cluster C, then Size of the Cluster is $\sum_{i=1}^{X} Size(Mod_i) \mid Mod_i \in C$

Balance criterion: To avoid having all cells migrate to one block, a balancing criterion is maintained. A partition (A, B) is balanced if,

$$r \times |V| - s_{max} \leq r \times |V| + s_{max}$$

where $|A| + |B| = |V|$ and $s_{max} = Max\{s(i)|1 \leq i \leq n\}$.

Algorithm 4.3.2 (DoPartition Algorithm)
DoPartition*(SourcePart, SubPart1, SubPart2)*
/ The function* **DoPartition***, takes input partition as first argument - SourcePart, and outputs two subpartitions as second and third arguments - SubPart1 and SubPart2. TL and BR are top left and bottom right coordinates, respectively, of partition */*

```
{     Cluster(SourcePart, SubPart1, SubPart2);
      $Ratio \leftarrow \mathbf{FM}(SourcePart, SubPart1, SubPart2)$;
      $Size \leftarrow \sum_{i=1}^{n} Size(Mod_i) \mid Mod_i \in Source\_Part$;
      $Size' \leftarrow (\frac{Ratio*100}{Size})$;
      $x \leftarrow Source\_Part(TL.X)$;
      $y \leftarrow Source\_Part(TL.Y)$;
      /* Upadate $Grid_{xy}$ */
      $i \mid 0 \leq i \leq Size'$
      {     if($Grid_{xy} \neq Source\_Part$) continue;
            $Grid_{xy} \leftarrow SubPart1$;
            $y \leftarrow y + 1$;
            if($y > Source\_Part(BR.Y)$)
            {     $x \leftarrow x + 1$;
            }     $y \leftarrow Source\_Part(TL.Y)$;
      }
      $\forall Grid_{xy} \mid Grid_{xy} = SourcePart\{Grid_{xy} \leftarrow SubPart2\}$;
      TerminalPropagation(SubPart1);
}     TerminalPropagation(SubPpart2);
```

Algorithm 4.3.3 (Cardinality Algorithm)
Cardinality*(SubPart)*

```
{     $\forall M_i \mid M_i \in SubPart \; \{Count \leftarrow Count + 1\}$;
      return Count;
}
```

Algorithm 4.3.4 (PutInTo_Cut Algorithm)

```
PutInTo_Cut(SubPart1, SubPart2)
{   $Count \leftarrow \textbf{Cardinality}(SubPart1)$;
    if $Count \geq 4$ $\texttt{To\_Cut} \leftarrow \texttt{To\_Cut} + SubPart1$;
        else $Legal\_Place \leftarrow Legal\_Place + SubPart1$;
    $Count \leftarrow \textbf{Cardinality}(SubPart2)$;
    if $Count \geq 4$ $\texttt{To\_Cut} \leftarrow \texttt{To\_Cut} + SubPart2$;
        else $Legal\_Place \leftarrow Legal\_Place + SubPart2$;
}
```

Connectivity of cell: Number of connections a cell has with the cluster being filled, is called the connectivity $Connectivity(i)$ of the cell i. It denotes the reduction in cutset due to the movement of cell i.

Cutset of partition: The *cutset* of a partition is the total number of nets connecting modules in the partition with macros outside the partition.

We perform clustering to form initial partition for FM bipartitioning algorithm. We extract two clusters out of the initial macro set `Source_Part`. Each contains macros which are densely connected in `Source_Part`. These clusters are presented into FM partitioning algorithm to obtain min cut bipartition. Since initial clusters are further operated by partitioner to refine the cutset, speed takes precedence over quality during clustering. Therefore, a simple greedy method is used to obtain the clusters.

To start, we unlock all the *pseudo modules (refer terminal propagation 4.3.6)* falling on cutline. Then place the modules locked to a position in appropriate cluster. We divide the area of FPGA allocated to the original macro set to be partitioned (`Source_Part`) between two clusters `DestPart1` and `DestPart2`. Then assign the modules locked to a position to the cluster, which is allocated to the area containing the locked module.

Then, we initialize `DestPart1` by an arbitrary macro from `Source_Part`. For all macros in `Source_Part`, their connectivity with respect to `DestPart1` is updated, and one with highest *connectivity* is placed in `DestPart1`. We repeat this process until *balance criterion* is satisfied. Then place all macros left over in `Source_Part`, in the other cluster `DestPart2`. `DestPart1` and `DestPart2` are given to FM for refining the cut set.

FM partitioning algorithm, being iterative in nature, is highly dependent on quality of initial cut. Hence a good initial cut produced by clustering, remarkably improves the performance of FM. Value of cutsets in first bipartitioning by FM for various input designs with random initial cut [5], and that with clustered initial cut is tabulated in the table 4.2.

4.3.5 FM Partitioner

It is the *Fiduccia - Mattheyses* algorithm implemented with slight modifications to accommodate IO pins, and zero sized pseudo modules introduced by *terminal propagation*. Zero sized modules are put only with

[5] Random cut was generated by alternately allocating a macro to each partition segment, till one of the partition segment was filled. Following which, all other macros were assigned to the other partition.

Design	Cutset by FM Partitioner	
	Random Initial Cut	Initial Cut With Clustering
CKT1	62	40
Mult16	79	45
MultRace	57	50
CLA	148	82
CPU	58	73

Table 4.2: *Cutset After First Cut on Input Design*

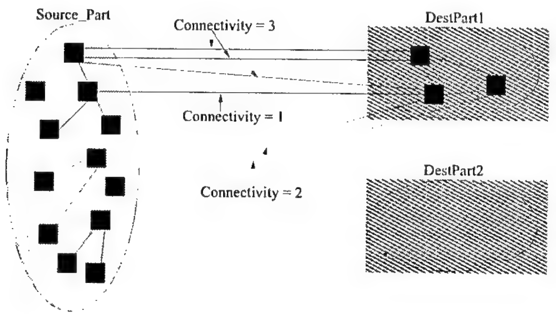


Figure 4.9: *An example to illustrate clustering*

connectivity as a constraint, not size, therefore are placed after placement of finite sized modules is completed.

The algorithm repeatedly tries to move the cell with the largest gain from its partition to the other partition provided the balance criterion is met. An Efficient data structure called a *bucket array* is used to implement the algorithm. Each entry in the array is an array of buckets with one bucket for every possible gain value. If p_{max} is the maximum number of pins connected to any cell, the gain value can vary from p_{max} to $-p_{max}$ and thus there are $2 * p_{max} + 1$ number of buckets.

Two bucket arrays (one for each partition) are maintained. Initially, in each pass, the cells are placed into the bucket corresponding to their gains. Whenever a cell i is chosen for moving, the corresponding entry from its bucket is removed. The gains of the cells connected to cell i are updated and the entries of these cells in the bucket array are updated based on the new gains.

At the end of a pass, the best sequence of moves are made permanent. The whole procedure is repeated for a few passes and the algorithm terminates when the cutsize does not change between consecutive passes.

4.3.6 Terminal Propagation

Following definitions are significant with respect to terminal propagation :

Pseudo Module : *Pseudo Module* is a module of size zero, introduced by the process of *terminal propagation.* A pseudo module is always contained in a *partition segment.*

Coordinates of Partition Segment : *Coordinates* of a partition segment are the coordinates of the area slice, to which it is mapped. They are represented by $Part.TL$ (top left coordinates) and $Part.BR$ (Bottom right coordinate). Top left coordinate of the FPGA chip is $(0, 0)$.

Algorithm 4.3.5 (Clustering Algorithm)
Cluster(Source_Part)

```
{       /* Unlock pseudo modules falling on cutline */
        if(Cut = Horizontal);
        {   $CutLine.Y \leftarrow \frac{Source\_Part.Top_y + Source\_Part.Bottom_y}{2}$;
            $\forall M_i \mid 1 \leq i \leq n; M_i \in Source\_Part;\ Lock(M_i).Y = CutLine.Y;\ Type(M_i) = PSEUDO$
                    $\{LockStatus(M_i) \leftarrow UNLOCKED\}$
        }
        else
        {   $CutLine.X \leftarrow \frac{Source\_Part.Left_x + Source\_Part.Right_x}{2}$;
            $\forall M_i \mid 1 \leq i \leq n; M_i \in Source\_Part;\ Lock(M_i).X = CutLine.X;\ Type(M_i) = PSEUDO$
                    $\{LockStatus(M_i) \leftarrow UNLOCKED\}$
        }
        /*Place locked modules in proper cluster*/
        $\forall M_i \mid 1 \leq i \leq n; M_i \in Source\_Part;\ Lock(M_i) \in DestPart_1$
                $\{DestPart_1 \leftarrow DestPart_1 + M_i\}$;
        $for\ each\ i \mid 1 \leq i \leq n;\ M_i \in DestPart_1\ \{Compute\ Connectivity(i)\}$;
        $Size_{Source} \leftarrow \sum_{i=1}^{n} size(M_i)$;
        $\forall M_i \mid M_i \in DestPart_1\ \{Size_{DestPart1} \leftarrow Size_{DestPart1} + size(M_i)\}$;
        /* Make first cluster DestPart1 */
        while($Size_{DestPart1} < (Size_{Source} * r)$)
        {   $NewModule \leftarrow M_i \mid i \leftarrow \max(Connectivity(i))$;
            $DestPart_1 \leftarrow DestPart_1 + NewModule$;
            $\forall M_i \mid M_i \notin DestPart_1; 1 \leq i \leq n\{Update\ Connectivity(M_i)\}$;
            $Size_{DestPart1} \leftarrow Size_{DestPart1} + size(NewModule)$;
        }
        /*Make Second cluster DestPart2*/
        $\forall M_i \mid M_i \notin DestPart_1; 1 \leq i \leq n\{DestPart_2 \leftarrow DestPart_2 + M_i\}$;
}
```

Algorithm 4.3.6 (Fiduccia-Mattheyses algorithm)

```
FM()
{
    Store unlocked zero size modules in a list;

    while()
    {
        $pass \leftarrow 1$;
        foreach $i$, $1 \leq i \leq n$, compute gain($g_i$)
        $i \leftarrow 1$;
        free cell set $F \leftarrow C$;
        while($F \neq \phi$)
        {
            base cell $b(i) \leftarrow j$, $j$ is the unlocked cell with maximum gain and choosing
            $j$ does not violate the balance criterion;
            If no base cell found break
            $m_i \leftarrow$ move cell $b(i)$ from its partition to the other partition;
            lock cell $b(i)$; $F \leftarrow F - b(i)$;
            update gains of all cells connected to cell $b(i)$;
            $i \leftarrow i + 1$;
        }
        select best sequence of moves $m_1, m_2 \dots m_k (1 \leq k \leq i)$ such that $G = \sum_{j=1}^{k} g_j$ is
        maximum;
        if $G < 0$ break
        make all $k$ moves ($m_1, m_2 \dots m_k$) permanent;
        calculate $cutsize_{pass}$;
        if $cutsize_{pass} = cutsize_{pass-1}$ break
        $pass \leftarrow pass + 1$;
    }
    Put Zero size modules in partitions according to their connectivity;
}
```

Lock(M_i) : It gives the position, to which the pseudo module M_i is locked. It is computed from the coordinates of its *parent partition segment.*

Pair of Pseudo Modules : Pseudo modules are introduced when a net is cut as a result of bipartitioning. One module each is introduced in both the partition segments. These modules are locked to adjacent positions on the boundary of the partition segments, and form a *pair*. *Pair* of pseudo modules always maintain their relative position.

Purpose of terminal propagation is to give knowledge of inter-partition connections to modules in each partition. This helps in reducing total interconnection length of the placed design. We begin with preprocessing the existing pseudo terminals. The pseudo terminals, which were unlocked during *clustering* (refer section 4.3.4, now belong to one of the subpartitions generated by *FM partitioner* (refer section 4.3.5), but are still unlocked. In this preprocessing, we first lock such pseudo modules. If a pseudo module belongs to a partition segment `Part` , its previous lock position was (x, y), then with current cutline as horizontal, its new lock position is $(x, \frac{Part.BR.Y+Part.TL.Y}{2})$, and with a vertical cutline it is $(\frac{Part.BR.X+Part.TL.X}{2}, y)$. After allocating the new lock position to all the unlocked pseudo modules, we lock the corresponding pseudo module in every module *pair*, such that the relative lock position of the *pair* of modules is maintained.

After this preprocessing, we introduce a new pseudo module for each cut-net in both the sub-partitions on either side of center of the cut-line. Hence, a new pair of pseudo modules belonging to two different partitions but locked to adjacent positions is introduced. Hereafter, whenever any pseudo module of this pair is moved to other location, the other pseudo module is also moved to maintain the same relative position. This forms an intermediate connection which makes the modules connected to these pseudo modules, in the corresponding partitions, to remain close to each other, as depicted in Figure 4.10.

In the figure 4.10, the modules shown M_a and M_b are cut by the first vertical cut V_1. At this point, two pseudo modules (shown as circles) are introduced and locked to the the center of the cutline V_1. Suppose, after the horizontal cut H_1, the module M_a gets placed in the top partition. Because of this movement, the pseudo module in the partition of M_1 moves to the center position on the previous vertical cut-line in the new partition of M_1. This makes the corresponding pseudo module of the pair to get attracted to the adjacent position. If the pseudo modules were not introduced and locked to the position as shown, the module M_b could have gone in the top or bottom partition made by horizontal cut H_2 without any bias to remain close to the module M_a. But as a result of the pseudo modules, M_b has a bias to go to the top partition, which will reduce the cutset by one, hence remain close to M_a. For the same reason, the two modules will tend to remain in the nearby partitions after further cuts too. Each time a cut is introduced, such pseudo modules are introduced and *weight of the net* is increased. This assures that modules connected to already cut nets remain close together, and also same net is not cut many times, hence minimizing the length of the net.

4.4 Legal Placement

At the end of successive bipartitioning, each leaf node of the partitioning tree contains maximum of K macros. Also, each leaf node is mapped on the target FPGA chip. In effect, it gives a locality on the FPGA chip corresponding to a leaf node, within which, macros contained in the leaf node should be placed. Legal placement is performed individually on each leaf node to decide the exact location and shape of macros contained in them. First, we decide relative placement of macros inside each partition. Then, we place hard macros. Finally, we process soft macros and place them. Processing of soft macros include reshaping, and deciding CLB placement inside the reshaped macro. Legal placement is also largely responsible for highly

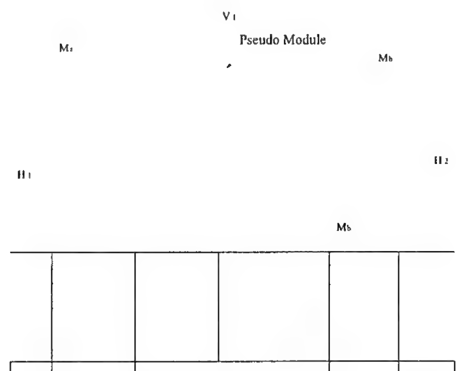


Figure 4.10: *Terminal Propagation & Pseudo Modules*

compact floorplan. This is achieved during reshaping and placement of soft macros. In this step, we place the macros keeping actual rectilinear boundaries of the modules in consideration, figure 4.11.

4.4.1 Relative placement of Macro Blocks

To obtain relative placement of macros inside a partition, we perform an exhaustive search for best relative placement (least total wirelength). There being maximum of K modules, maximum of $K!$ combinations exist in this search. Experimentally we found $K = 3$ to be good value. This is because, increasing the value of K does not improve the quality of the floorplan, but greatly increases the time taken for the *legal placement* step. On the other hand, reducing the value of K also results in increase in time required for floorplanning because of larger number of iterations required in successive bipartitioning step.

$$NumberofModules \leq K(K = 3)$$
$$MaxNumberofpasses \rightarrow K!$$

Pseudo modules, introduced during *terminal propagation (section 4.3.6)*, are still present at the boundaries of the partition, giving the direction from which each net enters or exits the partition. While deciding the relative placement of modules inside a partition, we account for wirelength for connections among various macros as well as the pseudo modules in the partition. This takes care of both inter-partition, and intra-partition connections. Hence an attempt to achieve global minimum wirelength is made.

To identify best arrangement out of all possible permutations of macros inside a partition, we compute number of connections between macros and those between macros and edges of partition. Connections on left edge of the partition are denoted by `LeftConnection` and those on right edge are denoted by `RightConnection`. Then, we calculate number of connections between each module in partition and `LeftConnection`, and store them in an array $Left_i$. We also calculate, number of connections between each module and `RightConnection`, and store them in an array $Right_i$. If maximum of $Left_i$ (i = P) is greater than maximum of $Right_i$ (i = Q), we place module M_P next to the left edge, and make `LeftConnection` a union of `LeftConnection` and pins on the modules placed. Otherwise, we place module M_Q next to right edge, and make `RightConnection` union of `RightConnection` and pins on module just placed. An array $Order_i$ with range 1 to k is used to store the sequence of modules. Index 1 indicates position next to left edge, and k indicates position next to right edge. We repeat this process until all the modules in the partition are placed.

Algorithm 4.3.7 (Terminal Propagation algorithm)

```
TP(Part1, Part2)
/* Part1 and Part2 are subpartitions created by last cut */
{       /* Lock all the unlocked pseudo modules */

        if(CutEdge = VERTICAL)
        {
            $X = \frac{(Part1.TL.X + Part1.BR.X)}{2};$
            $Y = Part1.BR.Y;$
            $\forall M_i \mid M_i \in Part_1; Type(M_i) = PSEUDO; LockStatus(M_i) = UNLOCKED$
              $\{Lock(M_i) \leftarrow (X,Y); Lock(M_{i+1}) \leftarrow (X,Y+1); LockStatus(M_i) = LOCKED\}$
        }
        else
        {
            $X = Part1.BR.X;$
            $Y = \frac{Part1.TL.Y + Part1.BR.Y}{2};$
            $\forall M_i \mid M_i \in Part_1; Type(M_i) = PSEUDO; LockStatus(M_i) = UNLOCKED$
              $\{Lock(M_i) \leftarrow (X,Y); Lock(M_{i+1}) \leftarrow (X+1,Y); LockStatus(M_i) = LOCKED\}$
        }

        /* Introduce new pseudo modules */

        if (CutEdge = VERTICAL)
        {
            $X = Part1.BR.X;$
            $Y = \frac{Part1.TL.Y + Part1.BR.Y}{2};$
            foreach net in CutSet
                IntroducePseudoMods(X, Y, X+1, Y);
        }
        else
        {
            $X = \frac{(Part1.TL.X + Part1.BR.X)}{2};$
            $Y = Part1.BR.Y;$
            foreach net in CutSet
                IntroducePseudoMods(X, Y, X, Y+1);
        }
}
```

Algorithm 4.3.8 (Introduce Pseudo Module algorithm)
IntroPseudoMods(X_1, Y_1, X_2, Y_2)
/ SubPart1 and SubPart2 are subpartitions created by last cut */*

```
{       $N \leftarrow NumModule + 1$;

        /* NumModules is Number of modules currently present */

        Introduce two new modules $M_N and M_{N+1}$;

        $Lock(M_N) \leftarrow (X_1, Y_1)$;

        $Lock(M_{N+1}) \leftarrow (X_2, Y_2)$;

        $LockStatus(M_N) \leftarrow LOCKED$;

        $LockStatus(M_{N+1}) \leftarrow LOCKED$;

        $Type(M_N) \leftarrow PSEUDO$;

        $Type(M_{N+1}) \leftarrow PSEUDO$;

        $NumModule \leftarrow NumModule + 2$;

        $Part_1 \leftarrow Part_1 + M_N$;

        $Part_2 \leftarrow Part_2 + M_{N+1}$;

        $NumModule \leftarrow NumModule + 2$;
}
```

Algorithm 4.4.1 (Relative place algorithm)

```
RP()
{   Foreach Partition
    {   $index \leftarrow 1$;
        $LeftConnection \leftarrow Connections\ on\ Left\ edge\ of\ Partition$;
        $RightConnection \leftarrow Connections\ on\ Right\ edge\ of\ Partition$;
        $\forall\ M_i \mid M_i \in Partition\ \{Left_i \leftarrow Connections\ between\ M_i\ and\ LeftConnection\}$
        $\forall\ M_i \mid M_i \in Partition\ \{Right_i \leftarrow Connections\ between\ M_i\ and\ RightConnection\}$
        $P \leftarrow \mathbf{Max}(left_i)\quad 1 \le i \le Number\ of\ Modules\ in\ Partition$
        $Q \leftarrow \mathbf{Max}(right_i)\quad 1 \le i \le Number\ of\ Modules\ in\ Partition$
        $\forall M_i \mid 1 \le i \le Number\ of\ Modules\ in\ the\ Partition$
        {
            if($P > Q$)
            {   $Order_{index} \leftarrow M_P$;
                $Partition \leftarrow Partition - M_P$;
                $LeftConnection \leftarrow LeftConnection + Pins\ on\ M_P$;
                $\forall M_i \mid M_i \in Partition$
                  $\{Left_i \leftarrow Connections\ between\ M_i\ and\ LeftConnection\}$
            }
            else
            {   $Order_{k-index+1} \leftarrow M_Q$;
                $Partition \leftarrow Partition - M_Q$;
                $RightConnection \leftarrow RightConnection + Pins\ on\ M_Q$;
                $\forall M_i \mid M_i \in Partition$
                  $\{Right_i \leftarrow Connections\ between\ M_i\ and\ RightConnection\}$
            }
            $index \leftarrow index + 1$;
        }
    }
}
```

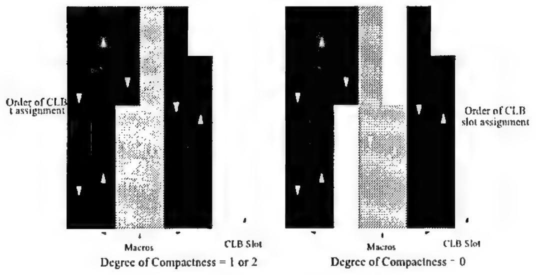

Order of CLB slot assignment
Macros
CLB Slot
Degree of Compactness = 1 or 2
Macros
CLB Slot
Degree of Compactness = 0
Order of CLB slot assignment

Figure 4.11: *Legal Placement : Non rectangular shapes of macros*

4.4.2 Place Hard Macros

Hard macros are a collection of CLBs with predetermined relative placement. Therefore, they have fixed dimensions. We did *successive bipartitioning*(section 4.3.4), and *relative placement*(section 4.4) considering module size only, not module dimensions. This may result in a situation, where in a partition, despite having enough area, dimension of hard macros may be too long, or too wide causing it not to fit in the space allocated to its parent *leaf cell* of the *partition tree*. In case of such a discrepancy, we place the macro in free space of adequate dimensions, nearest to the location allocated to it.

Suppose the decided module location is (x, y). If this location can not accommodate the hard macro M_i, then hard macro is placed at a location $(x + \delta x, y + \delta y)$, where $|\delta x|$ and $|\delta y|$ are minimum, and location $(x + \delta x, y + \delta y)$ can accommodate the hard macro module M_i.

Hence dimensions of hard macros are respected with minimum variation in its relative position decided in the floorplan.

4.4.3 Reshape and Place Soft Macros

Like hard macros, soft macros are also a collection of CLBs. Also, soft macros have fixed number of CLBs, hence fixed area. But they do not have pre determined relative locations of CLBs, hence do not have fixed dimensions. CLBs of soft macros are required to be mapped on the available CLB slots on the FPGA. We reshape soft macros to fit into the dimensions of the space allocated to their parent *leaf cell* in the *partition tree*. This is done by sequentially allocating available CLB slots on the FPGA to macros in the *partition segment*. This allocation is done in snake like fashion (even columns start from top row, and odd from bottom). Method of CLB assignment is such that, first we fill column of available space beginning from first row of the column. After a column is finished *col* + 1 is filled starting from last row up to the first row. Remaining CLBs of the macro start occupying the available CLB slots in next column. This process is continued until all CLBs inside the macro are placed. If *degree of compactness* desired is greater than zero, next soft macro starts where the previous one ends. Hence the shape of the soft macros can turn out to be non rectangular, which helps in placing the macros in minimum possible space (Figure 4.11). But if desired *degree of compactness* is zero, then next macro starts from top row of next column, in which case, shape of macros is always rectangular.

4.4.4 Simulated Annealing for Intra Macro Placement

CLBs inside each soft macro are placed using simulated annealing. Constraints are minimum total wire-length, and minimum longest wire. In this step also, both intra module connections, and inter module

Algorithm 4.4.2 (Reshape-place algorithm)

```
Reshape-Place()
{   foreach module $M_i$ in the Partition Part
    {   $Y = Part.Top_y$;
        $X = Part.Left_x$;
        $Size = M_i.Size$;
        $Grid_{XY} = M_i$;
        if($Row = EVEN$)
        {   $Y = Y + 1$;
            if($Y > Part.Bottom_y$)
            {   $X = X + 1$;
                $Y = Part.Bottom_y$;
                $Row = ODD$;
        }   }
        else
        {   $Y = Y - 1$;
            if($Y < Part.Top_y$)
            {   $X = X + 1$;
                $Y = Part.Top_y$;
                $Row = EVEN$;
        }   }
        $Size = Size - 1$;
        if($Size = \phi$)
        {   if($Compactness = \phi$)
            {   $Y = Part.Top_y$;
        }   }   $X = X + 1$;
    }
}
```

Algorithm 4.4.3 (Simulated Annealing)

```
SA()
{   $temp \leftarrow TEMP_{INIT}$;
    $place \leftarrow PLACE_{INIT}$;
    while($temp > TEMP_{FINAL}$)
    {   while($innerloopcriterion = FALSE$)
        {   $place_{new} \leftarrow PERTURB(place)$;
            if $place_{new}$ = Invalid Continue;
            $\delta C \leftarrow COST(place_{new}) - COST(place)$;
            if($\delta C < \phi$)
            {   $place \leftarrow place_{new}$;
            }
            else if($RANDOM(0,1) > e^{\frac{\delta C}{T}}$)
            {   $place \leftarrow place_{new}$;
            }
            $temp \leftarrow SCHEDULE(temp)$;
        }
    }
}
```

connections are considered. Hence a global picture is in view while attempting wirelength minimization.

The total wirelength is the sum of length for each net. The wirelength of a net is estimated by bounding box model.

For combinational circuits, the timing constraint is calculated as the sum of slack on the critical paths. The slack calculation for a given path is based on *path delay* analysis. A slack at an output IO pin IO_i is defined as,

$$s_i = r_i - a_i, \quad i \in \{1, 2, \ldots, p\}$$

where r_i is the latest required time at an output pad IO_i, a_i is the actual arrival time at an output pad IO_i and s_i is the path slack. A negative value for $s_i, \forall i$ indicates a violation of the timing constraint A path is deemed *critical* if its slack is negative.

Correspondingly for sequential circuits the slack is given by,

$$s_j = \frac{1}{f} - T^j_{comb}, \quad \forall j$$

where f is the expected clocking frequency, T^j_{comb} is the long path delay for a sub-path j where sub-path is a path between two clocked elements or a path between an IO pad and a clocked element. Also,

$$\max_j(s_j)$$

gives the maximum slack for the circuit and is the determining factor for the operational frequency of the circuit. The arrival time at the primary inputs is assumed to be zero.

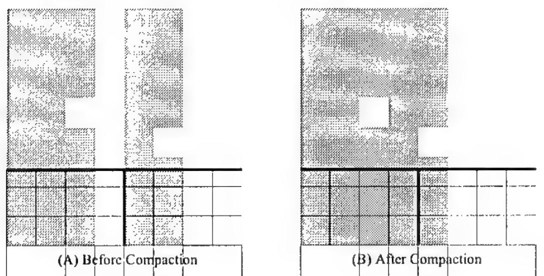


Figure 4.12: *Compaction : Degree of Compactness = 2*

4.5 Compaction

At the end of *legal placement (section 4.4)*, we arrive at a valid floorplan. This floorplan has tightly placed modules inside each *leaf cell* of the *partition tree*. But there are some unused CLBs surrounding the placed area inside *leaf cells*. This is illustrated in figure 4.12(A). Effectively, we have very tightly placed groups of macros separated by unused CLBs over the FPGA area. During compaction we work on floorplan generated by *legal placement* and strive to eliminate such unused space from within the bounding box of the floorplanned layout, without destroying relative placement of macros. Compaction produces a globally compact floorplan. We perform compaction by eliminating un-utilized rows and columns of CLBs on FPGA (if any) inside the boundary of the placed design. Partially empty rows and/or columns are left as they are. This ensures that relative placement obtained so far is respected.

The floorplanner can be executed with different *degrees of compaction*. Following is the effect *degree of compactness* on the final floorplanned layout :

- *Two* : Most compact floorplan is produced. Actual rectilinear boundaries are considered as described in *legal placement* step. Global compaction, as described in this section, is also done.

- *One* : As before, actual rectilinear boundaries of each macro is considered in *legal placement*, but *compaction* as described in this section is not performed. This leaves un-utilized spaces between *partition segments* hence providing extra resources to route.

- *Zero* : This produces least compact floorlan. Instead of considering actual rectilinear boundaries of each macro, *legal placement* is done by bounding boxes of macros (refer section 4.4.3). Also, *compaction* described in this section is not done. This leaves extra un-utilized spaces between macros, hence providing even more resources to route.

This step is carried out only if *degree of compactness* desired is 2. We identify rows of CLB on FPGA, which are completely unoccupied in the floorplan generated by *legal placement*. We move all modules up by one row(Y coordinate of their starting position - $M_i.Top_y$ - is decreased by 1), for each such row above them. Similarly, we identify empty columns, and move all the modules one column left (X coordinate of their starting position - $M_i.Left_x$ - is decreased by 1), for each such column to their left. Figure 4.12 illustrates an example floorplan before and after compaction.

At the end of *compaction*, we arrive at the final floorplan of the input design.

Algorithm 4.5.1 (algorithm)
Compact*()*

```
{       for all $row,\ \ 0 \leq row \leq Number\ of\ rows\ in\ FPGA$
            { if (row = empty)
                {$\forall M_i \mid M_i.Top_y > row, 1 \leq i \leq Number\ of\ modules\ in\ design$
                {$M_i.Top_y \leftarrow M_i.Top_y - 1$}}};
        for all $col,\ \ 0 \leq col \leq Number\ of\ columns\ in\ FPGA$
            { if (col = empty)
                {$\forall M_i \mid M_i.Left_x) > col, 1 \leq i \leq Number\ of\ modules\ in\ design$
                {$M_i.Left_x \leftarrow M_i.Left_x - 1$}}};
}
```

4.6 Routability

Routability of a circuit mapped on FPGA depends on two factors :

- *Available routing resources* : Total number of resources required for routing should not be more than the routing resources available on the FPGA.
- *Local Congestions* : If a particular locality of FPGA has very high density of logic mapped on it, some wire in the locality might not get routed because of the lack of resources there, despite availability of resources in other parts of the FPGA die. This situation arises when logic is densely packed in a locality, resulting in congestion.

First situation has been improved by keeping the average wirelength small. Successive bipartitioning with effective terminal propagation helps in reducing not only the maximum wirelength, but also the average wirelength of the design.

Second factor has been taken care of by providing several user controllable levels of compaction. Higher compaction level results in higher degree of logic packing, but can thwart routability for some dense circuits. Whereas lower compaction levels leaves some un-utilized space in the floorplan, thereby reducing the logic density, but providing more routing resources to prevent local congestion.

Chapter 5

Conclusions and Future Directions

5.1 Concluding Remarks

The RACE program resulted in several notable accomplishments. These include two architecture designs and their implementations and a software environment supporting the hardware architectures. It is a system level design project and we are pleased that almost everything worked without any glitch. RACE-I and NEBULA architectures have been used to demonstrate acceleration of various applications that are commonly used in DoD and commercial world. These include,

- Various filters for signal and image processing.
- Large scale arithmetic operations - multipliers.
- Discrete cosine transforms
- Fault tolerance of electronic circuits

Technology developed under RACE program is tranferrable to commercial sector and we are making progress in that direction. There is active communication with some potential points of interests. We intend to keep the program manager informed about any and every development related to technology transfer.

There are several areas in which more research and improvements can be made. CAD tools for reconfigurable computers are starting to emerge and are becoming stable. Execution time for physical design tools (place and route) still remains a bottlneck. Especially, with FPGA density growing at an enormous rate, the physical design tools have to address the challenges to break the computing time barrier. Our current work addresses these challenges.

I/O for reconfigurable computers is another performance bottleneck. Most reconfigurable computers are co-processors. RACE-I and NEBULA are also co-processors. The interface between main system bus and co-processor is thru a 16 bit or 32 bit wide bus. Advantages to computing on custom hardware are lost due to high time required to access the data using slow interface. Some of the new FPGA architectures like VIRTEX from Xilinx are addressing these issues by embedding small memory blocks within the FPGA and also by creating custom high speed interface between FPGA and external memory. Embedded processor cores with reconfigurable logic is a better solution. Our current research direction is to arrive at a system that embeds reconfigurable logic within the processor. Preliminary work in this direction has already started.

Bibliography

[1] AMD. An Introduction to Xilinx Products. Technical report, Xilinx Corporation, http://www.xilinx.com, 1997.

[2] XC6200 Field Programmable Gate Arrays. 1997.

[3] P. M. Athanas and H. F. Silverman. Processor Reconfiguration through Instruction-Set Metamorphosis. *IEEE Computer*, 26(3):11–18, March 1993.

[4] ATMEL. Found at the following web site: http://www.atmel.com/.

[5] S. Casselman. Virtual Computing and the Virtual Computer. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 43–48, Napa, CA, April 1993.

[6] Pak K. Chan. A Field-Programmable Prototyping Board: XC4000 BORG User's Guide. Technical Report UCSC-CRL-94-18, University of California at Santa Cruz, April 1994.

[7] Pak K. Chan, Martine D.F. Schlag, and Marcelo Martin. BORG: A Reconfigurable Prototyping Board using Field-Programmable Gate Arrays. In *1st International ACM/SIGDA Workshop on Field Programmable Gate Arrays*, pages 47–51, 1992.

[8] Charles E. Cox and W. Ekkehard Blanz. GANGLION—A Fast Field-Programmable Gate Array Implementation of a Connectionist Classifier. *IEEE Journal of Solid-state Circuits*, 27(3):288–299, March 1992.

[9] Steven A. Cuccaro and Craig F. Reese. The CM-2X: A Hybrid CM-2 / Xilinx Prototype. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 121–130, 1993.

[10] David E. Van den Bout, Joseph N. Morris, Douglas Thomae, Scott Labrozzi, Scot Wingo, and Dean Hallman. AnyBoard: An FPGA-Based, Reconfigurable System. *IEEE Design and Test of Computers*, pages 21–29, September 1992.

[11] Carl Ebeling and Neil McKenzie. Mactester: A Low-Cost Functional Tester for Interactive Testing and Debugging. In *3rd Microelectronics System Education Conference*, San Jose, CA, June 1990.

[12] D. Galloway. The Transmogrifier C Hardware Description Language and Compiler for FPGAs. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 136–144, Napa, CA, April 1995.

[13] P.G. Gulak. Field-Programmable Analog Arrays: A Status Report. *The 4th Canadian Workshop on Field-Programmable Devices*, pages 138–141, May 1996.

[14] J. D. Hadley and B. L. Hutchings. Designing a partially reconfigured system. In J. Schewel, editor, *Proceedings of the International Society of Optical Engineering (SPIE). Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing.*, volume 2607, pages 210–220, Philadephia, PA, October 1995.

[15] J.D. Hadley and B.L. Hutchings. Design Methodologies for Partially Reconfigured Systems. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 78–84, 1995.

[16] H. Högl, A. Kugel, J. Ludvig , R. Männer, Noffz K.-H., and R. Zoz. Enable++: A Second Generation FPGA Processor. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 45–53, Napa, CA, April 1995.

[17] B. L. Hutchings and M. J. Wirthlin. Implementation Approaches for Reconfigurable Logic Applications. In W. Moore and W. Luk, editors, *Field-Programmable Logic and Applications*, pages 419–428, Oxford, England, August 1995. Springer.

[18] J. W. Lockwood, *et al.* Scalable Optoelectronic ATM Networks: The iPOINT Fully Functional Testbed. *IEEE Journal of Lightwave Technology*, pages 1093–1103, June 1995.

[19] LSI Logic. *L64853A Enhanced SBus DMA Controller Technical Manual.* LSI Logic Corporation, 2nd edition, 1993.

[20] Lund Institute of Technology. SeeD-ROM. Found at the following web site: http://galaxy-3.dit.lth.se/~dsk/uppgift_e.html.

[21] Tomlinson G. Rauscher and Ashok K. Agrawala. Dynamic Problem-Oriented Redefinition of Computer Architecture via Microprogramming. *IEEE Transactions on Computers*, c-27(11):1006–1014, November 1978.

[22] Doug Smith and Dinesh Bhatia. RACE: Reconfigurable and Adaptive Computing Environment. In R. W. Hartenstein and M. Glesner, editors, *6th International Workshop on Field-Programmable Logic and Applications*, pages 87–95, Darmstadt, Germany, September 1996.

[23] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh. PRISM-II Compiler and Architecture. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 9–16, Napa, California, April 1993.

[24] Xilinx. *The Programmable Logic Data Book.* Xilinx, Inc., 1994.

[25] Xilinx. An Introduction to Xilinx Products. Technical report, Xilinx Corporation, http://www.xilinx.com, 1997.

[26] Xilinx. *LogiCORE PCI Master and Slave Interface User's Guide.* July 1997.

[27] Xilinx. XILINX: XC6200 Field Programmable Gate Arrays. Technical report, Xilinx Corporation, http://www.xilinx.com, 1997.

[28] Chengjin Zhang, Adrian Bratt, and Ian Macbeth. A New Field Programmable Mixed-Signal Array and Its Applications. *The 4th Canadian Workshop on Field-Programmable Devices*, pages 132–137, May 1996.